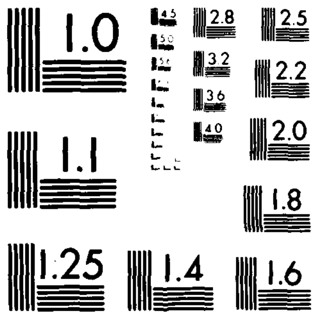
AD-A111 550 KENTUCKY UNIV LEXINGTON DEPT OF ELECTRICAL ENGINEERING F/G 9/2
INFERENTIAL PROCESSOR.(U)
JAN 82 F M BROWN, D K TAYLOR AFOSR-81-0115
UNCLASSIFIED AFOSR-TR-82-0037 NL

1 of 2
AD
A111550



MICROCOPY RESOLUTION TEST CHART

11

INFERENTIAL PROCESSOR

FINAL REPORT

for the period 1 April to 30 November, 1981
under Grant AFOSR 81-0115
Air Force Office of Scientific Research

Frank M. Brown
Donald K. Taylor

Department of Electrical Engineering
University of Kentucky
Lexington, Kentucky
40506-0046

January 1982

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 82 -0037	2. GOVT ACCESSION NO. AD111 550	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) INFERENTIAL PROCESSOR		5. TYPE OF REPORT & PERIOD COVERED FINAL, 1 APR 81-30 NOV 81
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Frank M. Brown and Donald K. Taylor		8. CONTRACT OR GRANT NUMBER(s) AFOSR-81-0115
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Electrical Engineering University of Kentucky Lexington KY 40506		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61102F; 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Mathematical & Information Sciences Directorate Air Force Office of Scientific Research Bolling AFB DC 20332		12. REPORT DATE JAN 1982
		13. NUMBER OF PAGES 128
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Results are presented concerning the design and application of an inferential processor, a digital machine organized to process logical data at high rates of speed. When coupled to a general-purpose computer the inferential processor would enable reasoning tasks to be carried out rapidly and with little programming effort. Specific research-efforts discussed in this report are (a) mechanized inference in Boolean systems, (b) functional deduction, and (c) inferential analysis of relational databases.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

INFERENTIAL PROCESSOR

FINAL REPORT

for the period 1 April to 30 November, 1981
under Grant AFOSR-81-0115
Air Force Office of Scientific Research

Frank M. Brown

Donald K. Taylor

Department of Electrical Engineering
University of Kentucky
Lexington, Kentucky
40506

January, 1982

PART A

STUDIES ON INFERENTIAL PROCESSING

F. M. Brown

FOREWORD

This Final Report presents the results of an eight-month project on the design and application of an **inferential processor**. The work on this project, conducted under Grant AFOSR 81-0115, commenced on 1 April 1981 and was completed on 30 November, 1981.

The research was carried out in the Department of Electrical Engineering at the University of Kentucky. Those principally involved were F. M. Brown (principal investigator), D. K. Taylor, and M. R. Rowlette; the latter two are graduate students who were supported by funds provided by the University of Kentucky.

ABSTRACT

Results are presented concerning the design and application of an inferential processor, a digital machine organized to process logical data at high rates of speed. When coupled to a general-purpose digital computer, the inferential processor would enable reasoning tasks to be carried out rapidly and with little programming effort. Specific research-efforts discussed in this report are (a) mechanized inference in Boolean systems, (b) functional deduction, and (c) inferential analysis of relational databases.

TABLE OF CONTENTS

I. INTRODUCTION

The objective of the research described in this report has been to investigate the design and application of an *inferential processor*, a machine specialized for rapid processing of Boolean (i.e., propositional) data. This research is part of a longer-term effort to mechanize a new approach to reasoning in propositional logic. The basic ideas underlying this approach have been worked out over a period of several years; the practical implementation of those ideas was first undertaken in 1980, however, while the principal investigator was at the Air Force Avionics Laboratory under the sponsorship of the USAF/SCEEE Summer Faculty Research Program.

The proposed *inferential processor*, which is intended to augment the computational power of a general-purpose computer, is to be a high-speed reasoning system having very general capability within the domain of propositional logic. It may be implemented either by microprogramming a general-purpose computer or by attaching to such a computer a special-purpose processor; the latter implementation [1] is assumed in this report.

Our research during the grant-period has been organized into the following tasks:

1. Mechanized Inference in Boolean Systems (F.M. Brown);
2. Functional Deduction (F.M. Brown);
3. Boolean Analysis of Relational Databases (D.K. Taylor);
4. Simulation of the Inferential Processor (M.R. Rowlette).

The foregoing research-tasks were undertaken as eight-month efforts promising the greatest progress toward the objectives stated in our proposal. We present in this report the results of the first three tasks; the results of the final task are to appear in an M.S. thesis which is currently underway.

The objective of the first task, Mechanized Inference in Boolean Systems, was to develop an organized and coherent theory of Boolean analysis. The basic inferential operations on systems of Boolean equations were studied, terminology was established, and properties fundamental to the operation of the inferential processor were proved. The objects of the first task were principally those of clarification, terminology, and proof. In the second task, Functional Deduction, our object was to investigate a new application of the processor, one which had only been sketched in our previous research [2]. We believe functional deduction to be a fundamental operation in Boolean analysis; it is the inverse, essentially, of the much-studied problem of solving Boolean equations. The results obtained under this task enable functional deduction to be performed rapidly and efficiently by the inferential processor. To study its essential features and illustrate its practical utility, we have applied functional deduction to the design of economical multiple-output combinational circuits.

The objective of the third task, Boolean Analysis of Relational Databases, was to investigate potential applications of the inferential processor to database processing. We began by studying the problems associated with relational databases. This study showed that the generation of keys for a database is a difficult problem of practical importance. The keys may be determined if the functional dependencies associated with the database are known; we therefore devised an algorithm (the first to our knowledge) for generating the functional dependencies in a given relational database. This algorithm also produces the full set of minimal keys for the database. The algorithm was programmed entirely in the logical language PROLOG [3,4], which was used for two reasons: first, this language is most effective for programming tasks involving logic; second, PROLOG is an "inferential processor" in software, whose operation we wished to study.

This report is organized in two parts. Part A includes general background on logical computers and some discussion of the motivation for our research (Section II), a brief description of the structure of the inferential processor (Section III), and discussions of the results obtained under tasks 1 and 2 above (Sections IV and V). Part B, originally prepared as an M.S. thesis [5], presents the results obtained under task 3.

References

1. Brown, F. M., "Inferential Processor," Final Report, AFOSR/SCEEE Summer Faculty Research Program, August 1980.

2. Brown, F. M., "High-Speed Reasoning in Propositional Logic," Proposal to AF Office of Scientific Research, July 1981.

3. Roussel, P., "PROLOG: manuel de reference et d'utilization," Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille, Luminy, France, September 1975.

4. Clocksin, W.F. and C.S. Mellish, Programming in Prolog. N.Y.: Springer-Verlag, 1981.

5. Taylor, D.K., "Analyzing Relational Databases Using Propositional Logic," M. S. Thesis, Department of Electrical Engineering, University of Kentucky, December, 1981.

II. LOGICAL COMPUTERS

We outline in this section the motivation for our research, whose ultimate object is to produce a logical computer, i.e., a machine capable of high-speed inferential processing in propositional (Boolean) logic. Some of the material in the present section is taken from a proposal [1] prepared during the grant-period; it is included in this report for completeness.

Propositional logic may be identified roughly with two-valued Boolean algebra. This form of logic has applications in many areas, a few of which are logical design [2], the diagnosis of failures in digital systems [3,4], and the design of relational databases [5]. It is the basis, moreover, for reasoning in higher-order logics such as the first-order predicate calculus; the latter is required for applications in artificial intelligence [6,7]. The propositional calculus is related to the higher-order logics in somewhat the same way that arithmetic is related to the various fields of mathematical analysis; it is a structure, useful in itself, on which more elaborate structures are built.

The range of application of the propositional calculus was outlined by Ledley [8] as follows: "The propositional calculus can be applied to many phases of military science and related problems as well as to business, industry, science, and government in general. In these applications it serves as an aid to complex reasoning, e.g., in the analysis and evaluation of intelligence reports, the preparation and analysis of tactical methods and principles, the formulation and interpretation of legal statutes, the planning and evaluation of chemical and biological experiments, the formulation of psychological and intelligence examinations, and the formulation and evaluation of business methods and procedures. All of these and similar 'reasoning' activities and operations can use the propositional calculus in a fundamental way. More well-known are its applications to the design of industrial process-control machines, digital computers, large-scale switching circuitry, and other forms of information-handling systems. However, the computational methods of the propositional calculus present serious and frequently insurmountable difficulties in the solution of actual problems, and this factor has severely limited its practical utilization. Consequently the need arises for a systematic way of formulating, analyzing and solving propositional functions and equations."

Notwithstanding the "logical" nature of its internal operations, a general-purpose computer is ill-suited to logical computation. For this reason, a number of dedicated logical processors have been proposed. For a detailed study of logic-machines, from the *Ars Magna* of Ramon Lull in the thirteenth century to the relay-machines of the 1950's, see Gardner [9]. The electronic machines relevant to the present project may be put into three classes: *argument-verifiers*, *equation-solvers*, and *formula-minimizers*.

All of the argument-verifiers [10-16] known to this investigator (and none of the other kinds of logical computers) have been designed by logicians. The function of any such machine is to decide the validity of an argument, i.e., a collection of premises together with a conclusion. Equation-solving machines [17-22], on the other hand, have been inspired principally by the need to solve technological problems. Such machines accept some representation of a system of Boolean equations and produce a solution (typically particular rather than general) for a selected subset of the arguments in terms of the remaining arguments.

Formula-minimizing machines [23-27] have the common aim of determining simplified sum-of-products (disjunctive normal form) expressions for propositional functions. The procedures implemented in all of these machines are based on Quine's formulation [28-30] of the minimization problem, the essential feature of which is the generation of the *prime implicants* of the given propositional function. Formula-minimization may at first glance appear to have little bearing on mechanized inference. It is significant, however, for two reasons. First, formula-minimization is useful in the application of other reasoning processes, improving the economy and perspicuity of the results obtained. Second, the existing designs for formula-minimizing machines represent solutions of a problem that is dominant in the design of the proposed inferential processor, namely, that of generating and storing the prime implicants of a propositional function.

Each of the machines cited above carries out a species of reasoning; each extracts useful information, that is, from a collection of propositional data. None of these machines has emerged from the laboratory of its birth, however, because none is in any sense "general-purpose" within the domain of propositional logic. The element absent in these machines is a *central principle of reasoning*, readily adaptable to argument-verification, equation-solving, formula-minimization, and any other task involving logical inference. The proposed inferential processor embodies such a principle, viz., that the *prime consequences* characterize, in a simple and economical way, all conclusions deducible from a collection of propositional data.

The technique of automated inference we are investigating is based on a formulation given by A. Blake in a little-known dissertation [31] published in 1937. The concept of a prime implicant, customarily attributed to a paper published by Quine [28] fourteen years later, as well as all of the presently-known methods for generating prime implicants, were presented in Blake's dissertation. The application of prime implicants to formula-minimization was pointed out by Quine and has since been intensively studied and applied; Blake's application of prime implicants to logical deduction, however, has apparently remained unnoticed. Blake's principal contribution was to show that a single rule of inference, that of Hypothetical Syllogism (if P implies Q and Q implies R, then P implies R), suffices to produce all of the prime consequences of a collection of propositional data. Expressed in terms of Boolean algebra, the single operation of consensus (which Blake called the "syllogistic result") suffices to produce all of the prime implicants of a Boolean function. This idea is closely related to the "resolution principle" given by Robinson [32] in 1965 and now applied in mechanical theorem- proving [6] and in programming languages, such as PROLOG [33,34], designed to solve problems in the predicate calculus. When compared with Blake's use of consensus, the resolution principle is formulated in a more general structure (the first-order predicate calculus) and is applied to a less general problem (theorem-proving by refutation).

Blake demonstrated the fundamental role of the prime consequences in generating and verifying conclusions. Our research has shown an additional advantage of the prime consequences, namely, that they enable the fundamental operations of propositional inference (e.g., elimination of variables, solution of equations in general and particular form, general of functional antecedents and consequents) to be conveniently mechanized in a high-speed processor. Thus a machine that accepts propositional data and produces their prime consequences can be made general-purpose in the domain of propositional logic.

References

1. Brown, F. M., "High-Speed Reasoning in Propositional Logic," proposal submitted to AF Office of Scientific Research, July, 1981.

2. Shannon, C. E., "A symbolic analysis of relay and switching circuits," Trans. AIEE, vol. 57, pp. 713-723, 1938.

3. Bossen, D. C. and S. J. Hong, "Cause-effect analysis for multiple fault detection in combinational networks," IEEE Trans. on Computers, vol. C-20, pp. 1252-1257, Nov. 1971.

4. Gatlin, A. B. and M. A. Fal'kovich, "Diagnosis of memoryless digital automata using a modified technique of natural logical deduction," Automation and Remote Control (USSR), vol. 41, no. 1, part 2, pp. 104-109, January 1980.

5. Sagiv, Y., et al., "An equivalence between relational database dependencies and a fragment of propositional logic," to appear in the Journal of the Association for Computing Machinery.

6. Chang, C. L. and R. C. T. Lee, Symbolic Logic and Mechanical Theorem Proving. N.Y.: Academic Press, 1973.

7. Nilsson,N. J., Problem-Solving Methods in Artificial Intelligence. N.Y.: McGraw-Hill, 1971.

8. Ledley, R. S., "Mathematical foundations and computational methods for a digital logic machine," Journal of the Operations Research Society of America, vol. 2, no. 3, pp. 249-274, August 1954.

9. Gardner, M., Logic Machines and Diagrams. N.Y.: McGraw-Hill, 1958.

10. Bauer, F. L., "The formula-controlled logical computer STANISLAUS," Math. Computation, vol. 14, pp. 64-67, Jan. 1960.

11. Burack, B., "An electrical logic machine," Science, vol. 109, pp. 610-611, 17 June 1949.

12. Burks, A. W., et al., "An analysis of a logical machine using parenthesis-free notation," Mathematical Tables and Other Aids to Computation, vol. 8, pp. 53-57, April 1954.

13. Couffignal, Louis, "Sur un probleme d'analyse mecanique abstraite," C.R. Acad. Sci. Paris, vol. 206, pp. 1336-1338, 1938.

14. Marquand, A., "A new logical machine," Proc. Amer. Academy of Arts and Sciences, vol. 21, p. 303, 1885.

15. Mays, W. and D. G. Prinz, "A relay machine for the demonstration of symbolic logic," Nature, vol. 165, p. 197, Feb. 4, 1950.

16. Miehle, W., "Burroughs truth function evaluator," Journal of the Association for Computing Machinery, pp. 189-192, April, 1957.

17. Cherry, E. C. and P. K. T. Vaswani, "A new type of computer for problems in propositional logic, with greatly reduced scanning procedures," Information and Control, vol. 4, p. 155-168, 1961.

18. Marin, M. A., "Investigation of the field of problems for

the Boolean Analyzer," Dep't. of Electrical Engineering Report no. 68-28, Univ. of Calif. at Los Angeles, 1968.

19. McCallum, D. M. and J. B. Smith, "Mechanized reasoning: logical computers and their design," Electronic Engineering, vol. 23, pp. 126-133, April 1951.

20. Rose, A., Computer Logic. N.Y.: Wiley-Interscience, 1971. Chapter 3, "Logical Computers."

21. Svoboda, A., "Boolean Analyzer," Proc. IFIP Congress, Edinburgh, 1968, pp. 824-830.

22. Toon, W. M., "Microprogrammed Boolean Analyzer," M.S. Thesis, Dep't. of Electrical Engineering, Univ. of Kentucky, December, 1978.

23. Florine, J., "Optimization of binary functions with a special-purpose electronic computer," Automation and Remote Control (USSR), vol. 28, no. 6, pp. 956-962, June, 1967.

24. Garton, R. D., "A hardware realization of a decomposition algorithm," Report R-511, Coordinated Science Laboratory, University of Illinois, May 1971.

25. Gerace, G. B. et al., "TOPI--a special-purpose computer for Boolean analysis and synthesis," IEEE Trans. on Computers, vol. C-20, no. 8, pp. 837-842, August 1971.

26. Gomez-Gonzalez, L., "Estudio teorico, concepcion y realizacion de un sistema electronico para simplificar funciones logicas," Ph.D. Thesis, Dpto. Electricidad y Electronica, Facultad de Ciencias, Universidad de Granada, Spain, 1977.

27. Matney, R. M. and C. H. Roth, "Associative computing structures for the solution of logic design problems," SWIEEE Conference Record, 1969.

28. Quine, W. V., "The problem of simplifying truth functions," Am. Math. Monthly, vol. 59, no. 8, pp. 521-531, Oct. 1952.

29. Quine, W. V., "A way to simplify truth functions," Am. Math. Monthly, vol. 62, no. 10, pp. 627-631, Nov. 1955.

30. Quine, W. V., "On cores and prime implicants of truth functions," Am. Math. Monthly, vol. 66, no. 9, pp. 755-760, November 1959.

31. Blake, A., Canonical Expressions in Boolean Algebra. Chicago: University of Chicago Libraries, 1938 (Reprint of Ph.D. Dissertation, Department of Mathematics, University of Chicago, 1937).

32. Robinson, J. A., "A machine-oriented logic based on the resolution principle," Journal of the Association for Computing Machinery, vol. 12, no. 1, pp. 23-41, January 1965.

33. Roussel, P., "PROLOG: manuel de reference et d'utilisation," Groupe d'Intelligence Artificielle, Universite d'Aix-Marseille, Luminy, France, September, 1975.

34. Kowalski, R., *Logic for Problem Solving*. N.Y.: North-Holland, 1979.

III. ORGANIZATION OF THE INFERENTIAL PROCESSOR

We present in this section a brief outline of the organization of the proposed inferential processor; see [1] for a more complete description.

The function of the inferential processor is to accept, store, and process Boolean or propositional data. It is intended to function as a high-speed adjunct to a general-purpose computer, as indicated in Figure 1.

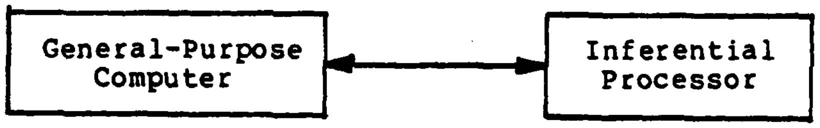


Fig. 1. Total system.

The applications anticipated for the inferential processor fall into two main classes: (a) tasks involving only propositional (Boolean) logic and (b) tasks involving higher-order logic, primarily the first-order predicate calculus. The first class includes such applications as computer-aided design of logic-circuits [2], the design and analysis of databases (see [3], which is included as Part B of this report), and on-line diagnosis of faults in digital systems [4]. In the second class of applications, unlike the first, the inferential processor does not do all of the logical work; instead, it provides high-speed subroutines for use by the general-purpose computer. The employment of the inferential processor in the latter class of applications is based on the fact that higher-order logics employ propositional logic as their basic "arithmetic." Most applications of this class come under the heading of artificial intelligence, many branches of which depend heavily on the first-order predicate calculus.

Principal Components

The major components of the inferential processor are shown in Figure 2.

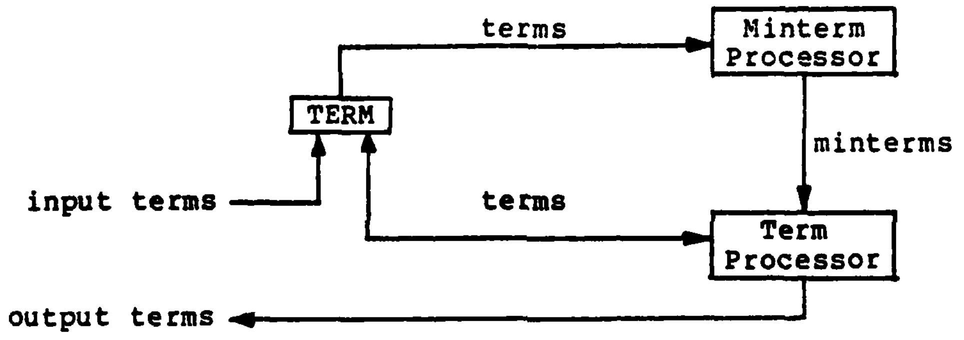


Fig. 2. Major components of the inferential processor.

The unit labelled TERM is a register that holds the term (Boolean product) currently under consideration. The Minterm Processor accepts terms from TERM, building from them a Boolean function $F(x_1,\ldots,x_n)$ using AND, OR, NOT, EOR, etc. The function F is represented in the Minterm Processor by its minterm canonical form. The Term Processor accepts the minterms of F from the Term Processor and generates the Blake canonical form, i.e., the disjunction of the prime implicants, of F. The Term Processor carries out the fundamental operations of logical analysis (elimination of variables, solution of equations, etc.) which, arranged in programmed sequences, carry out the processing requested by the general-purpose computer.

Major Phases of Operation

The operation of the inferential processor takes place in three major phases: reduction, development, and analysis. The reduction-phase, carried out in the Minterm Processor, reduces a system of logical equations to a single equation having the form F = 0. The development-phase, carried out in the Term Processor, generates a representation of F in Blake canonical form. The analysis-phase, carried out in both processors, executes the sequence of inferential operations requested by the general-purpose computer.

References

1. Brown, F. M., "Inferential Processor," Final Report, AFOSR/SCEEE Summer Faculty Research Program, August, 1980.
2. Svoboda, A. and D. E. White, *Advanced Logical Circuit Design Techniques*. N.Y.: Garland STPM Press, 1979.
3. Taylor, D. K., "Analyzing Relational Databases using Propositional Logic," M.S. Thesis, Department of Electrical Engineering, University of Kentucky, December, 1981.
4. Brown, F. M. and H. H. Yeh, "Boolean Equations and Logical Diagnosis," submitted to IEEE Transactions on Computers.

IV. MECHANIZED INFERENCE IN BOOLEAN SYSTEMS

The task of the proposed inferential processor is to accept logical data in the form of a Boolean system and to generate useful inferences (conclusions) from such a system. We have attempted in this project to develop a systematic formulation of (a) the properties of Boolean systems and (b) the principal operations on such systems that are of use in logical inference. We discuss that formulation in this section.

Review of Elementary Properties

The equivalences

$$a \leq b \iff a\bar{b} = 0 \tag{1}$$

$$a = b \iff a \oplus b = 0 \tag{2}$$

$$[a = 0 \quad \text{and} \quad b = 0] \iff a + b = 0 \tag{3}$$

$$[a = 1 \quad \text{and} \quad b = 1] \iff ab = 1 \tag{4}$$

are valid for arbitrary elements a and b in a Boolean algebra. Equivalences (3) and (4) have obvious extensions to more than three variables; thus $[a = 0 \text{ and } b = 0 \text{ and } c = 0]$ is equivalent to $[a + b + c = 0]$, etc.

Boolean Systems

An n-variable Boolean system on a Boolean algebra B is a collection

$$\begin{aligned} g_1(\underline{x}) &= h_1(\underline{x}) \\ &\vdots \\ g_k(\underline{x}) &= h_k(\underline{x}) \\ g_{k+1}(\underline{x}) &\leq h_{k+1}(\underline{x}) \\ &\vdots \\ g_m(\underline{x}) &\leq h_m(\underline{x}) \end{aligned} \tag{5}$$

of simultaneously-asserted equations and inclusions in which the g's and h's are n-variable Boolean functions on B and $\underline{x}$ denotes the vector $(x_1, \ldots, x_n)$.

The number, k, of equations may be zero in a system, as may the number, m, of inclusions; we require, of course, that there be at least one equation or one inclusion in a Boolean system.

Solutions. An element $\underline{b}$ of B^n is a solution of the system (5) if each of the statements in (5) becomes an identity under the substitution $\underline{x} = \underline{b}$. A Boolean system is said to be consistent if it has at least one solution; otherwise, it is said to be inconsistent.

Implication and equivalence. Let S_1 and S_2 be two n-variable Boolean systems on B. We say that S_1 implies S_2, written $S_1 \Rightarrow S_2$, in case the statement

$$(\forall \underline{b} \in B^n)\ [\underline{b} \text{ is a solution of } S_1 \Rightarrow \underline{b} \text{ is a solution of } S_2]$$

is true. Note that S_1 implies any n-variable Boolean system if S_1 is inconsistent. We say that two Boolean systems S_1 and S_2 are equivalent, written $S_1 \Leftrightarrow S_2$, if each implies the other, i.e., if each has the same set of solutions. Any two inconsistent systems, in particular, are equivalent.

Reduction

By (1) and (2), the system (5) is equivalent to the system

$$\begin{aligned} g_1(\underline{x}) \oplus h_1(\underline{x}) &= 0 \\ &\vdots \\ g_k(\underline{x}) \oplus h_k(\underline{x}) &= 0 \\ g_{k+1}(\underline{x})\, \bar{h}_{k+1}(\underline{x}) &= 0 \\ &\vdots \\ g_m(\underline{x})\, \bar{h}_m(\underline{x}) &= 0 \quad . \end{aligned} \tag{6}$$

System (6) is equivalent, by (3), to the single equation

$$f(\underline{x}) = 0, \tag{7}$$

where f is a Boolean function defined by

$$f = \sum_{i=1}^{k} (g_i \oplus h_i) + \sum_{i=k+1}^{m} g_i \bar{h}_i \,. \tag{8}$$

By similar reasoning, invoking (4) instead of (3), we deduce that the system (5) is equivalent to the single equation

$$F(\underline{x}) = 1 \,, \tag{9}$$

where F is a Boolean function defined by

$$F = \prod_{i=1}^{k} (\bar{g}_i \oplus h_i) \cdot \prod_{i=k+1}^{m} (\bar{g}_i + h_i). \tag{10}$$

Any Boolean system can therefore be "boiled down" to a single equation of the form (7) or of the form (9). We will focus principally on the form (7).

<u>Example 1</u>. The system

$$ax = b + y$$
$$ab \leq a\bar{x} + \bar{y}$$

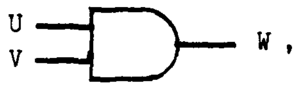
is equivalent to the system

$$a\bar{b}x\bar{y} + \bar{a}b + \bar{a}y + b\bar{x} + \bar{x}y = 0$$
$$ab(\bar{a}y + xy) = 0 \,,$$

which is equivalent, in turn, to the single equation

$$a\bar{b}x\bar{y} + \bar{a}b + \bar{a}y + b\bar{x} + \bar{x}y + abxy = 0.$$

Example 2. The behavior of an AND-gate,



is described by the three equivalent statements

$$UV = W \tag{11}$$

$$\bar{U}W + \bar{V}W + UV\bar{W} = 0 \tag{12}$$

$$\bar{U}\bar{W} + \bar{V}\bar{W} + UVW = 1 \quad . \tag{13}$$

Boolean Relations

Given a Boolean algebra B and a vector $\underline{x} = (x_1,\ldots,x_n)$, a relation (or constraint) on $\underline{x}$ is a statement that confines $\underline{x}$ to lie within a subset of B^n. The operation of the AND-gate of Example 2, for instance, is specified by the relation

$$(U,V,W) \in \{(0,0,0),(0,1,0),(1,0,0),(1,1,1)\} \;, \tag{14}$$

where $B = \{0,1\}$ and $\underline{x} = (U,V,W)$. (Strictly speaking, the relation is the subset $\{(0,0,0),(0,1,0),(1,0,0),(1,1,1)\}$ itself; it is convenient for our present purposes, however, to call the statement (14) a relation.)

Two relation-statements on $\underline{x} = (x_1,\ldots,x_n)$ will be called equivalent if they confine $\underline{x}$ to the same subset of B^n. Thus statement (14) above is equivalent to equation (11), as well as to equations (12) and (13).

An identity on $\underline{x} = (x_1,\ldots,x_n)$ is a relation equivalent to the statement

$$\underline{x} \in B^n.$$

An identity, in other words, is a relation on $\underline{x}$ that doesn't really "confine" $\underline{x}$ at all. The relations $\bar{x}_1x_2 \leq x_2$ and $x_1 + x_2 = x_1 + \bar{x}_1x_2$, for example, are both identities on (x_1,x_2).

A relation on $\underline{x} = (x_1,\dots,x_n)$ will be called a Boolean relation if it is equivalent to a Boolean equation, i.e., if it is equivalent to a statement of the form

$$f(\underline{x}) = 0 \ ,$$

where $f\colon B^n \to B$ is a Boolean function. If $B = \{0,1\}$, then every relation on $\underline{x}$ is a Boolean relation. If B is a Boolean algebra larger than $\{0,1\}$, then not all relations are Boolean. Suppose $B = \{0,1,\bar{a},a\}$. Then the relation

$$(x_1,x_2) \in \{(0,0),\ (a,0)\} \tag{15}$$

is a Boolean relation because it is equivalent to the Boolean equation

$$\bar{a}x_1 + x_2 = 0 \ . \tag{16}$$

The set $\{(0,0),\ (a,0)\}$ of solutions of (16), that is, is precisely the set defining the relation (15). The relation

$$(x_1,x_2) \in \{(0,0),\ (a,1)\} \ , \tag{17}$$

on the other hand, is not a Boolean relation. It is not equivalent, that is, to a Boolean equation; any Boolean equation $f(x_1,x_2) = 0$ on $B = \{0,1,\bar{a},a\}$ having solutions $(0,0)$ and $(a,1)$ must also have solutions $(0,\bar{a})$ and (a,a)--as we shall be able to show after we discuss the solution of Boolean equations.

Eliminants

Let $f\colon B^n \to B$ be a Boolean function expressed in terms of arguments $x_1,\dots,x_n$. We derive from f a set $\{C_T f \mid T \subseteq \{x_1,\dots,x_n\}\}$ of Boolean functions by applying the following rules:

(i) $$C_{\emptyset} f = f$$
$$C_{\{x_1\}} f = f(0,x_2,\dots,x_n)\cdot f(1,x_2,\dots,x_n)$$

(ii) $$C_{R \cup S} f = C_R(C_S f) \ .$$

We derive another set, $\{D_T f \mid T \subseteq \{x_1,\ldots,x_n\}\}$ of Boolean functions by applying the rules

$$\text{(i)} \qquad D_{\emptyset} f = f$$
$$D_{\{x_1\}} f = f(0,x_2,\ldots,x_n) + f(1,x_2,\ldots,x_n)$$
$$\text{(ii)} \qquad D_{R \cup S} f = D_R(D_S f) \; .$$

We call $C_T f$ the <u>conjunctive</u> <u>eliminant</u>, and $D_T f$ the <u>disjunctive</u> <u>eliminant</u>, of f with respect to the subset T of $\{x_1,\ldots,x_n\}$. Note that if x is a single letter, then the conjunctive and disjunctive eliminants of f with respect to x are related to the discriminants $f_{\bar{x}}$ and f_x (discussed in Chapter 4) as follows:

$$C_{\{x\}} f = f_{\bar{x}} \cdot f_x \tag{18a}$$
$$D_{\{x\}} f = f_{\bar{x}} + f_x \; . \tag{18b}$$

It is convenient to omit set-braces in specifying eliminants; thus we write $C_{x_1 x_3} f$ rather than $C_{\{x_1,x_3\}} f$.

Suppose the subset T comprises k elements ($k \leq n$) of $\{x_1,\ldots,x_n\}$ (we assume without loss of generality that T comprises the first k elements, i.e., that $T = \{x_1,\ldots,x_k\}$). Then $C_T f$ and $D_T f$ are determined as follows:

$$C_T f = \prod_{\underline{b} \in \{0,1\}^k} f(\underline{b}, x_{k+1},\ldots,x_n)$$
$$D_T f = \sum_{\underline{b} \in \{0,1\}^k} f(\underline{b}, x_{k+1},\ldots,x_n) \; .$$

If $k=2$ and $n=4$, for example, then

$$C_{wx} f(w,x,y,z) = f(0,0,y,z) \cdot f(0,1,y,z) \cdot f(1,0,y,z) \cdot f(1,1,y,z)$$

and

$$D_{wx} f(w,x,y,z) = f(0,0,y,z) + f(0,1,y,z) + f(1,0,y,z) + f(1,1,y,z) .$$

It is clear that the conjunctive and disjunctive eliminants of a Boolean function f with respect to a subset T of the argument-set may be expressed by formulas not involving any of the arguments in T. The process of calculating such formulas may in some cases be greatly simplified by application of the two theorems which follow.

Theorem 1. Let $f: B^n \rightarrow B$ be a Boolean function expressed in terms of arguments $x_1,\ldots,x_n$. Then

$$BCF(C_{x_1} f) = \sum (\text{terms of } BCF(f) \text{ not involving } \bar{x}_1 \text{ or } x_1).$$

Proof: The literals $\bar{x}_1$ and x_1 may be factored from the terms of BCF(f) in which they appear, in such a way that f is expressed as

$$f = \sum_{i=1}^{L} \bar{x}_1 p_i(x_2,\ldots,x_n) + \sum_{j=1}^{M} x_1 q_j(x_2,\ldots,x_n) + \sum_{k=1}^{N} r_k(x_2,\ldots,x_n),$$

where $p_1,\ldots,p_L,q_1,\ldots,q_M,r_1,\ldots,q_N$ are terms (products) not involving the argument x_1. Thus $C_{x_1} f = f(0,x_2,\ldots,x_n)f(1,x_2,\ldots,x_n)$ may be expressed as $\left[\sum_{i=1}^{L} p_i + \sum_{k=1}^{N} r_k\right]\left[\sum_{j=1}^{M} q_j + \sum_{k=1}^{N} r_k\right] = \sum_{i=1}^{L}\sum_{j=1}^{M} p_i q_j + \sum_{k=1}^{N} r_k$. Every consensus formed by terms of BCF(f) is absorbed by a term of BCF(f). In particular, every consensus of the form $p_i q_j$ is absorbed by one of the r-terms; thus $\sum_{i=1}^{L}\sum_{j=1}^{M} p_i q_j \leq \sum_{k=1}^{N} r_k$, and we conclude that $C_{x_1} f = \sum_{k=1}^{N} r_k$. Thus $C_{x_1} f$ may be expressed as the portion of BCF(f) that remains after every term involving $\bar{x}_1$ or x_1 is deleted. It is shown in Chapter 4 that the result of such deletion is in Blake canonical form ■

Corollary 1.1. Let $f: B^n \rightarrow B$ be a Boolean function expressed in terms of arguments $x_1,\ldots,x_n$ and let T be a subset of $\{x_1,\ldots,x_n\}$. Then

$$BCF(C_T f) = \sum (\text{terms of } BCF(f) \text{ not involving arguments in } T). \quad (19)$$

__Proof:__ By Theorem 1, (19) is valid if $\#T = 1$, i.e., if T is a singleton-set. Suppose (19) to be valid if $\#T = k$, and consider the case $\#T = k+1$, i.e., let $T = \{x_i\} \cup R$, where $\#R = k$ and $x_i \notin R$. Then $\mathrm{BCF}(C_T f) = \mathrm{BCF}(C_{\{x_i\}}(C_R f)) =$ $\sum$(terms of $\mathrm{BCF}(C_R f)$ not involving $\bar{x}_i$ or x_i) $= \sum$(terms of $\sum$(terms of BCF(f) not involving arguments in R) not involving $\bar{x}_i$ or x_i). Thus (19) is valid for $T = \{x_i\} \cup R$ ■

__Example 3.__ The system

$$\bar{w} + x = y$$

$$x + \bar{y} = wz$$

is equivalent to the single equation

$$x\bar{y} + w\bar{x}y + w\bar{z} + \bar{w}\bar{y} + \bar{y}\bar{z} + \bar{w}x + x\bar{z} = 0 \ ,$$

whose left side, f, is expressed in Blake canonical form. The conjunctive eliminants expressed below are constructed by inspection of BCF(f), using Theorem 1 and its corollary.

$$C_x f = w\bar{z} + \bar{w}\bar{y} + \bar{y}\bar{z} \qquad C_{wx} f = C_w(C_x f) = \bar{y}\bar{z}$$

$$C_w f = x\bar{y} + \bar{y}\bar{z} + x\bar{z} \qquad C_{wx} f = C_x(C_w f) = \bar{y}\bar{z} \ .$$

__Theorem 2.__ Let $f: B^n \longrightarrow B$ be a Boolean function expressed in terms of arguments $x_1, \ldots, x_n$. Then $D_{x_1} f$ is obtained from any SOP formula for f by replacing $\bar{x}_1$ and x_1, wherever they appear in the formula, by 1.

__Proof:__ By definition, $D_{x_1} f(x_1, x_2, \ldots, x_n) = f(0, x_2, \ldots, x_n) + f(1, x_2, \ldots, x_n)$. An SOP formula for f may be expanded in the form

$$f(x_1, x_2, \ldots, x_n) = \bar{x}_1 p(x_2, \ldots, x_n) + x_1 q(x_2, \ldots, x_n) + r(x_2, \ldots, x_n),$$

where p, q, and r are SOP formulas not involving x_1; hence, $f(0, x_2, \ldots, x_n) =$ $p(x_2, \ldots, x_n) + r(x_2, \ldots, x_n)$ and $f(1, x_2, \ldots, x_n) = q(x_2, \ldots, x_n) + r(x_2, \ldots, x_n)$.

We deduce, therefore, that

$$D_{x_1} f(x_1, x_2, \ldots, x_n) = p(x_2, \ldots, x_n) + q(x_2, \ldots, x_n) + r(x_2, \ldots, x_n).$$

Thus $D_{x_1} f$ is produced by replacing the literals $\bar{x}_1$ and x_1 by 1 in the original SOP formula for f $\blacksquare$

We refer to the foregoing procedure, which was given first apparently by Mitchell [3] , as the "replace-by-1 trick."

Example 4. Let f(w,x,y,z) be given by

$$f = \bar{w}\bar{x}y\bar{z} + \bar{w}xyz + w\bar{y}z .$$

Then

$$D_{wx} f = y\bar{z} + yz + \bar{y}z = y + z$$

$$D_{yz} f = \bar{w}\bar{x} + \bar{w}x + w = 1 .$$

Example 5. The following (correct) calculations illustrate potential pitfalls in applying the replace-by-1 trick:

(a) $D_u(\bar{u} + vw) = 1 + vw = 1$

(b) $D_u(\overline{u + v}) = D_u(\bar{u}\bar{v}) = \bar{v}$

(c) $D_u(u + v)(\bar{u} + w) = D_u(uw + \bar{u}v + vw) = w + v$

Calculation (a) illustrates that $D_u f$ is not found simply by deleting $\bar{u}$ and u (which would produce vw rather than 1 in this case), but by replacing $\bar{u}$ and u by 1. Calculations (b) and (c) illustrate the necessity that f be expressed in sum-of-products form before the literals $\bar{u}$ and u are replaced by 1. If the replacements are made in the original formulas, then the erroneous results would be $D_u(\overline{u + v}) = (\overline{1 + v}) = 0$ for (b) and $D_u(u + v)(\bar{u} + w) = (1 + v)(1 + w) = 1$ for (c).

Elimination

A Boolean relation constrains the vector $\underline{x} = (x_1, \ldots, x_n)$ to lie within a subset of B^n. It also constrains any k-element subvector of $\underline{x}$ $(1 \leq k \leq n)$ to lie within a subset of B^k. The Boolean relation (14) describing an AND-gate, for example, constrains (U,V,W) to lie within the subset $\{(0,0,0), (0,1,0), (1,0,0), (1,1,1)\}$ of the 8-element set $\{0,1\}^3$. Suppose we wish to find the implied relation on the subvector (U,W). To do so, we simply delete the middle element of each triple in (14), and keep the set of pairs that remains. The resulting relation on (U,W) is

$$(U,W) \in \{(0,0), (1,0), (1,1)\}. \tag{20}$$

We say in this case that V has been <u>eliminated</u> from the relation (14) to produce the relation (20), and we call (20) the <u>resultant of elimination</u> of V from (14). Relation (20), it should be emphasized, limits (U,W) to the same subset of $\{0,1\}^2$ as does the original relation (14).

If R is a Boolean relation, i.e., one equivalent to a Boolean equation

$$f(x_1, x_2, \ldots, x_n) = 0, \tag{21}$$

then the resultant of elimination of any argument from R is also a Boolean relation. Thus, the resultant of elimination of x_1 from the equation (21) may be expressed by an equation of the form

$$g(x_2, \ldots, x_n) = 0. \tag{22}$$

To determine the resultant (22) from equation (21), we may proceed by (i) expressing (21) as an equivalent explicit subset of B^n, (ii) deleting the first element of each n-tuple in the subset, and (iii) expressing the resulting subset of B^{n-1} as an equation of the form (22). The following result enables us, however, to generate (22) directly from (21).

<u>Theorem</u> 3. The equation $g(x_2,\ldots,x_n)=0$ expresses the resultant of elimination of x_1 from the equation $f(x_1,x_2,\ldots,x_n)=0$ if and only if the identity

$$g = C_{x_1} f \tag{23}$$

is fulfilled.

<u>Proof</u>: The fundamental theorem of Boolean algebra, together with properties (1) through (3), gives rise to the following chain of equivalences:

$$\begin{gathered}
f(x_1,x_2,\ldots,x_n) = 0 \\
\Updownarrow \\
\bar{x}_1 f(0,x_2,\ldots,x_n) + x_1 f(1,x_2,\ldots,x_n) = 0 \\
\Updownarrow \\
\begin{cases} \bar{x}_1 f(0,x_2,\ldots,x_n) = 0 \\ x_1 f(1,x_2,\ldots,x_n) = 0 \end{cases} \\
\Updownarrow \\
\begin{cases} f(0,x_2,\ldots,x_n) \leq x_1 \\ x_1 \leq \bar{f}(1,x_2,\ldots,x_n) \end{cases} \\
\Updownarrow \\
f(0,x_2,\ldots,x_n) \leq x_1 \leq \bar{f}(1,x_2,\ldots,x_n) \quad .
\end{gathered} \tag{24}$$

The constraint imposed by (10) on the subvector $(x_2,\ldots,x_n)$ is

$$f(0,x_2,\ldots,x_n) \leq \bar{f}(1,x_2,\ldots,x_n) \ ,$$

which may be re-expressed as

$$f(0,x_2,\ldots,x_n) \bullet f(1,x_2,\ldots,x_n) = 0.$$

Thus $g=0$ is the resultant of elimination of x_1 from $f=0$ if and only if the condition (23) is satisfied

<u>Corollary 3.1</u>. The equation $G(x_2,\ldots,x_n)=1$ expresses the resultant of elimination of x_1 from the equation $F(x_1,x_2,\ldots,x_n)=1$ if and only if the identity

$$G = D_{x_1} F \tag{25}$$

is fulfilled.

<u>Example 6</u>. The AND-gate discussed in Example 2 is characterized by either of the equations $f(U,V,W) = 0$ or $F(U,V,W) = 1$, where the functions f and F are defined by

$$\begin{aligned} f &= \bar{U}W + \bar{V}W + UV\bar{W} \\ F &= \bar{U}\bar{W} + \bar{V}\bar{W} + UVW \ . \end{aligned} \tag{26}$$

Applying Theorem 1 and its corollary, the resultant of elimination of V is expressed by either of the equations $g(U,W)=0$ or $G(U,W)=1$, where

$$\begin{aligned} g = C_V f &= f(U,0,W) \bullet f(U,1,W) \\ &= (\bar{U}W + W) \bullet (\bar{U}W + U\bar{W}) \\ &= \bar{U}W \end{aligned}$$

$$\begin{aligned} G = D_V F &= F(U,0,W) + F(U,1,W) \\ &= (\bar{U}\bar{W} + \bar{W}) + (\bar{U}\bar{W} + UW) \\ &= U + \bar{W} \ . \end{aligned}$$

The resultant is expressed, therefore, either by $\overline{U}W = 0$ or by $U + \overline{W} = 1$; either of these equations, or the equivalent inclusion $W \leq U$, is equivalent to the relation (20). These relations express all that is known concerning U and W, in the absence of knowledge concerning V.

If we eliminate W from (12), the resultant is $(UV)(\overline{U} + \overline{V}) = 0$, i.e., $0 = 0$. The latter relation on (U,V) is an identity; it allows (U,V) to be chosen freely, that is, from $\{0,1\}^2$--which confirms our expectation that the inputs to an AND-gate should be unconstrained in value if nothing is known concerning the value of the output.

The Extended Verification Theorem

We discuss in this section a result, due to Löwenheim [2] and Müller [4], which enables an implication between two Boolean equations to be translated into an equivalent Boolean inclusion. The presentation in this section is adapted from that of Rudeanu [6].

Let s be a single element of B and let $\underline{v} = (v_1, v_2, \ldots, v_n)$ be a vector on B, i.e., $s \in B$ and $\underline{v} \in B^n$. Then $s\underline{v}$ and $\underline{v}s$ are defined by

$$s\underline{v} = \underline{v}s = (sv_1, sv_2, \ldots, sv_n) .$$

Lemma 1. Let $f: B^n \rightarrow B$ be a Boolean function and let $\underline{b}$ be an element of B^n such that $f(\underline{b}) = 0$. Then

$$f(\underline{b}\,f(\underline{x}) + \underline{x}\,\overline{f}(\underline{x})) = 0 \qquad \forall\, \underline{x} \in B^n . \tag{27}$$

Proof: By the fundamental theorem of Boolean algebra,

$$f(\underline{b}\,f(\underline{x}) + \underline{x}\,\overline{f}(\underline{x})) = \overline{f}(\underline{x})\,f(\underline{x}) + f(\underline{x})\,f(\underline{b}) .$$

Each term on the right-hand side of the foregoing equation has the value zero, for any $\underline{x} \in B^n$, proving (27). ■

Theorem 4 (Extended Verification Theorem). Let $f: B^n \rightarrow B$ and $g: B^n \rightarrow B$ be Boolean functions, and assume that the equation $f(\underline{x}) = 0$ is consistent. Then the following statements are equivalent:

(i) $(\forall \underline{x} \in B^n)\ [f(\underline{x}) = 0 \Rightarrow g(\underline{x}) = 0]$

(ii) $(\forall \underline{x} \in B^n)\ [g(\underline{x}) \leq f(\underline{x})]$

(iii) $(\forall \underline{x} \in \{0,1\}^n)\ [g(\underline{x}) \leq f(\underline{x})]$.

Proof:

(i) $\Rightarrow$ (ii): Let $\underline{b} \in B^n$ be a solution of $f(\underline{x}) = 0$, i.e., let $f(\underline{b}) = 0$. Then $g(\underline{b}) = 0$. For any $\underline{x} \in B^n$, $f(\underline{x}\,\bar{f}(\underline{x}) + \underline{b}\,f(\underline{x})) = 0$ by Lemma 1; hence, $g(\underline{x}\,\bar{f}(\underline{x}) + \underline{b}\,f(\underline{x})) = 0$. Thus, for all $\underline{x} \in B^n$, $\bar{f}(\underline{x})\,g(\underline{x}) + f(\underline{x})\,g(\underline{b}) = \bar{f}(\underline{x})\,g(\underline{x}) = 0$, i.e., $g(\underline{x}) \leq f(\underline{x})$, proving (ii).

(ii) $\Rightarrow$ (iii): Trivial.

(iii) $\Rightarrow$ (i): The functions f and g are Boolean; hence, they may be written in minterm canonical form, i.e., $f(\underline{x}) = \sum_{i=0}^{2^n-1} f_i\, m_i(\underline{x})$ and $g(\underline{x}) = \sum_{i=0}^{2^n-1} g_i\, m_i(\underline{x})$ for all $\underline{x} \in B^n$. Assume (iii), i.e., assume that $g_i \leq f_i$ $(i = 0,1,\ldots,2^n-1)$, and let $\underline{b} \in B^n$ be a solution of $f(\underline{x}) = 0$. Then $\sum_{i=0}^{2^n-1} f_i\, m_i(\underline{b}) = 0$, which implies that $f_i\, m_i(\underline{b}) = 0$, and therefore that $g_i\, m_i(\underline{b}) = 0, (i = 0,1,\ldots,2^n-1)$. Thus $g(\underline{b}) = 0$, proving (i). ■

Corollary 4.1. Let $f: B^n \rightarrow B$ and $g: B^n \rightarrow B$ be Boolean functions and assume that the equation $f(\underline{x}) = 0$ is consistent. Then the following statements are equivalent:

(i) $(\forall \underline{x} \in B^n)\ [f(\underline{x}) = 0 \Leftrightarrow g(\underline{x}) = 0]$

(ii) $(\forall \underline{x} \in B^n)\ [f(\underline{x}) = g(\underline{x})]$

(iii) $(\forall \underline{x} \in \{0,1\}^n)\ [f(\underline{x}) = g(\underline{x})]$.

Proof: Immediate from Theorem 3 and the definition of equivalent systems. ■

Poretsky's Law of Forms

It is useful on some occasions to re-express the information supplied by the Boolean equation $f(\underline{x}) = 0$ in the equivalent form $g(\underline{x}) = h(\underline{x})$, where g is any given Boolean function. The associated Boolean function h is specified by the following theorem.

Theorem 5 (Poretsky's Law of Forms). Let $f,g,h\colon B^n \rightarrow B$ be Boolean functions and suppose the equation $f(\underline{x}) = 0$ to be consistent. Then the equivalence

$$f(\underline{x}) = 0 \iff g(\underline{x}) = h(\underline{x}) \tag{28}$$

holds for all $\underline{x} \in B^n$ if and only if

$$h = f \oplus g \; . \tag{29}$$

Proof: Suppose (28) to hold for all $\underline{x}$ in B^n. Then (28) is equivalent, by property (2) and Corollary 3.1, to the equation $f(\underline{x}) = g(\underline{x}) \oplus h(\underline{x}) \quad (\forall \underline{x} \in B^n)$. Thus $g(\underline{x}) \oplus f(\underline{x}) = g(\underline{x}) \oplus (g(\underline{x}) \oplus h(\underline{x})) = h(\underline{x}) \quad (\forall \underline{x} \in B^n)$, from which we deduce (29) directly. Suppose on the other hand that the function h is defined by (29). Let $\underline{b} \in B^n$ be one of the solutions of the consistent equation $f(\underline{x}) = 0$. Then $h(\underline{b}) = f(\underline{b}) \oplus g(\underline{b}) = 0 \oplus g(\underline{b}) = g(\underline{b})$, i.e., $\underline{b}$ is also a solution of $g(\underline{x}) = h(\underline{x})$ (and we deduce that $g(\underline{x}) = h(\underline{x})$ is consistent). Thus $f(\underline{x}) = 0 \implies g(\underline{x}) = h(\underline{x})$. To show that $g(\underline{x}) = h(\underline{x}) \implies f(\underline{x}) = 0$, let $\underline{c} \in B^n$ be any solution of $g(\underline{x}) = h(\underline{x})$. Then $g(\underline{c}) \oplus h(\underline{c}) = 0$, whence $g(\underline{c}) \oplus (f(\underline{c}) \oplus g(\underline{c})) = 0$ by (29), from which we deduce that $f(\underline{c}) = 0$, proving (28). ■

Example 7. Suppose a Boolean function h is sought having the property that the equation $x_1\bar{x}_2 + x_3 = 0$ is equivalent to $x_2x_3 = h$. The first equation is consistent (a solution, for example, is $x_1 = 0$, $x_2 = 0$, $x_3 = 0$); hence, h is determined uniquely by (29), i.e.,

$$h = (x_1\bar{x}_2 + x_3) \oplus (x_2x_3)$$
$$= \bar{x}_2(x_1 + x_3) .$$

References

1. Boole, G., An Investigation of the Laws of Thought. London: Walton, 1854.
2. Löwenheim, L., "Über die Auflösung von Gleichungen im logischen Gebietekalkul," Math. Ann., vol. 68, pp. 169-207, 1910.
3. Mitchell, O. H., "On a new algebra of logic," in Studies in Logic, ed. by C. S. Peirce. Boston: Little, Brown, and Co., 1883.
4. Müller, E., Abriss der Algebra der Logik. Leipzig, 1909-1910. Published as an appendix to vol. III of Ernst Schröder's Algebra der Logik (1890-1905) by Chelsea Pub. Co., New York, 1966.
5. Poretsky, P., "Sept lois fondamentales de la théorie des égalités logiques," Bulletin de la Societé Physico-Mathématique de Kasan, ser. 2, vol. 8, pp. 33-103, 129-181, 183-216, 1898.
6. Rudeanu, S., Boolean Functions and Equations. Amsterdam: North-Holland, 1974.

V. FUNCTIONAL DEDUCTION

An important potential application of the inferential processor is that of generating functional consequences, i.e., conclusions of the form

$$x_1 = f(x_2,\dots,x_n), \tag{30}$$

from a given system of logical equations on the variables $x_1, x_2, \dots x_n$. If such consequences exist, then we call x_1 functionally deducible from the given equations and we say that $\{x_2,\dots,x_n\}$ is a determining subset for x_1. Generating functional consequences from a given system of equations is the inverse of solving the system; if (30) is a solution of a system, then the system is a consequence of (30). The problem of solving logical equations was given primary attention in Boole's original work, and has since been studied intensively; there has been no progress to our knowledge, however, on the problem of generating functional consequences.

Some very preliminary work on functional deduction was reported in [1]; we outline in this section the progress we have made in the meantime. A test (Theorem 6) is given to determine, for a given logical database, the functionally deducible variables; this test is well-suited for high-speed execution by the inferential processor, inasmuch as it is based on the basic units of data (prime implicants) stored in the processor. Given that a variable is functionally deducible, the set of functions f for which (30) is a functional consequence is specified by Corollary 6.1. A necessary and sufficient condition for a subset of $\{x_2,\dots,x_n\}$ to be an x_1-determining subset is given in Theorem 7, and an algorithm is given to generate the class of minimal x_1-determining subsets. Finally, the theory of functional deduction is applied to the problem of designing economical multiple-output combinational circuits.

The discussion in this section is based on the concepts and terminology introduced in Section II.

Functional Deducibility

Let us suppose a collection of Boolean, i.e., propositional, data to be reduced by the inferential processor to the single equation

$$\phi(x_1,x_2,\ldots,x_n) = 1 . \tag{31}$$

We say that x_1 is functionally deducible from (31) in case there is a Boolean function $f:B^{n-1} \rightarrow B$ such that equation (30) is a consequence of (31). We call (30) a functional consequence of (31).

Theorem 6. The following statements are equivalent:

(i) x_1 is functionally deducible from $\phi(x_1,\ldots,x_n) = 1$.

(ii) $D_{x_1}\bar{\phi} = 1$.

(iii) $C_{x_1}\phi = 0$.

(iv) x_1 or $\bar{x}_1$ appears in every term of BCF(ϕ).

Proof:

(i)$\Longleftrightarrow$(ii)$\Longleftrightarrow$(iii): The equivalence of the following statements follows directly from the results of Section II. In particular, the equivalence of (a) and (b) follows from the extended verification theorem (Theorem 4) and property (2).

(a) $(\exists f)\ [\phi(x_1,\ldots,x_n) = 1 \implies x_1 = f(x_2,\ldots,x_n)]$

(b) $(\exists f)\ [\phi(x_1,x_2,\ldots) \leq \bar{x}_1 \oplus f(x_2,\ldots)]$

(c) $(\exists f) \begin{bmatrix} \phi(0,x_2,\ldots) \leq \bar{f}(x_2,\ldots) \\ \phi(1,x_2,\ldots) \leq f(x_2,\ldots) \end{bmatrix}$

(d) $(\exists f)\ [\phi(1,x_2,\dots) \leq f \leq \bar{\phi}(0,x_2,\dots)]$

(e) $\phi(1,x_2,\dots) \leq \bar{\phi}(0,x_2,\dots)$

(f) $\begin{bmatrix} \bar{\phi}(1,x_2,\dots) + \bar{\phi}(0,x_2,\dots) = 1 \\ \phi(1,x_2,\dots) \cdot \phi(0,x_2,\dots) = 0 \end{bmatrix}$

(g) $\begin{bmatrix} D_{x_1}\bar{\phi} = 1 \\ C_{x_1}\phi = 0 \end{bmatrix}$

(iii)$\Longleftrightarrow$(iv): The terms of BCF($C_{x_1}\phi$), by Theorem 1, are those of ϕ which do not involve $\bar{x}_1$ or x_1. Thus $C_{x_1}\phi = 0$ if and only if $\bar{x}_1$ or x_1 appears in every term of BCF(ϕ).

Corollary 6.1. Let $f: B^{n-1} \rightarrow B$ be a Boolean function. Then the equation $x_1 = f(x_2,\dots,x_n)$ is a functional consequence of $\phi(x_1,\dots,x_n) = 1$ if and only if the functions f and ϕ satisfy the condition

$$\phi_{x_1} \leq f \leq \bar{\phi}_{\bar{x}_1} \quad . \tag{32}$$

Minimal Determining Subsets

Let $\{\{u\},V,W\}$ be a partition of $\{x_1,\dots,x_n\}$, where $V = \{v_1,\dots,v_p\}$ and $W = \{w_1,\dots,w_q\}$. We say that V is a u-determining subset of $\{x_1,\dots,x_n\}$, and that W is u-eliminable, if u is deducible from the equation

$$D_W\phi = 1 \ . \tag{33}$$

Theorem 7 Let $\{\{u\},V,W\}$ be a partition, as described above, of the set $\{x_1,\dots,x_n\}$. Then V is a u-determining subset if and only if the condition

$$D_W(\phi_u) \cdot D_W(\phi_{\bar{u}}) = 0 \tag{34}$$

is satisfied, in which case a functional consequence is $u = f(\underline{v})$, where f is any Boolean function in the non-empty interval

$$D_W(\phi_u) \leq f \leq D_W(\phi_{\bar{u}}). \qquad (35)$$

Proof: By Theorem 6, the variable u is deducible from $D_W = 1$ if and only if $C_u(D_W\phi) = 0$, i.e., $(D_W\phi)_u \cdot (D_W\phi)_{\bar{u}} = 0$. From the identities

$$(D_W\phi)_u = D_W(\phi_u) \qquad (36a)$$

$$(D_W\phi)_{\bar{u}} = D_W(\phi_{\bar{u}}), \qquad (36b)$$

we conclude that u is deducible from $D_W\phi = 1$ if and only if (34) is satisfied, in which case, by Corollary 6.1, we obtain the equation $u = f(\underline{v})$ as a consequence, where $(D_W\phi)_u \leq f \leq \overline{(D_W\phi)_{\bar{u}}}$. Identities (36a) and (36b) lead therefore to (35).

Generating minimal determining subsets. The following procedure, based on Theorem 7, generates a convenient representation of the class of minimal u-determining subsets.

Step 1. Express ϕ_u and $\phi_{\bar{u}}$ as sum-of-products (disjunctive normal) formulas, viz.,

$$\phi_u = \sum_{i=1}^{m} p_i$$

$$\phi_{\bar{u}} = \sum_{j=1}^{n} q_j$$

Step 2. Associate with each pair (p_i, q_j) of terms an alterm s_{ij} defined by the summation

$$s_{ij} = \sum (\text{letters that appear opposed in } p_i \text{ and } q_j).$$

Step 3. Construct the product-of-sums formula

$$a_u = \prod_{i=1}^{m} \prod_{j=1}^{n} s_{ij} .$$

Step 4. Multiply out, to form a sum-of-products formula for a_u, and delete absorbed terms.

Step 5. With each term $xy\cdots z$ of a_u associate a minimal u-determining subset $\{x,y,\ldots,z\}$.

Example 8. Let us examine for functional deducibility the data given in a problem widely quoted by early logicians (Boole [2], Chapter IX):

"Suppose that an analysis of the properties of a particular class of substances leads to the following statements:

(1) Whenever properties A and C are missing, then property E is found, together with one of the properties B and D, but not both.

(2) Whenever the properties A and D are found while E is missing, then both B and C will either both be found or both be missing.

(3) Whenever property A is found in conjunction with either B or E, or both of them, then C or D will also be found, but not both of them. Conversely, whenever C or D (but not both) is found, then A will be found in conjunction with either B or E or both of them."

The foregoing data are equivalent to the single equation

$$\phi = 1, \tag{37}$$

where ϕ is given in Blake canonical form by

$$BCF(\phi) = \bar{A}CD + ABC\bar{D} + A\bar{C}DE + AC\bar{D}E + \bar{A}B\bar{C}\bar{D}E + A\bar{B}\bar{C}\bar{D}\bar{E}. \tag{38}$$

The variables appearing in every term of $BCF(\phi)$ are A, C, and D; hence, by Theorem 6, these are the variables functionally deducible from (37).

Let us consider the functionally deducible variable A; in particular, let us determine the minimal A-determining subsets of $\{B,C,D,E\}$.

$$\phi_A = BC\bar{D} + \bar{C}DE + C\bar{D}E + \bar{B}\bar{C}\bar{D}\bar{E}$$

$$\phi_{\bar{A}} = CD + B\bar{C}\bar{D}E$$

Thus,

$$\begin{aligned} a_A &= (D)(C)(C)(D)(D)(C)(C + D)(B + E) \\ &= CD(B + E) \\ &= BCD + CDE. \end{aligned}$$

The minimal A-determining subsets, therefore, are {B,C,D} and {C,D,E}. To determine f in the functional consequence A = f(B,C,D), we apply (35) in Theorem 7, viz.,

$$D_{\{E\}}(\phi_A) \leqslant f \leqslant \overline{D_{\{E\}}(\phi_{\bar{A}})}.$$

Thus,

$$BC\bar{D} + \bar{C}D + C\bar{D} + \bar{B}\bar{C}\bar{D} \leqslant f \leqslant C\bar{D} + \bar{B}\bar{C} + \bar{C}D.$$

Two simplified functional consequences are derived from the foregoing interval, viz.,

$$A = C\bar{D} + \bar{C}D + \bar{B}\bar{C}$$

$$A = C\bar{D} + \bar{C}D + \bar{B}\bar{D}.$$

Similar analysis yields the following functional consequence based on the A-determining subset V = {C,D,E}:

$$A = C\bar{D} + \bar{C}D + \bar{D}\bar{E}.$$

We noted earlier that the variables functionally deducible from (37), in addition to A, are C and D. The (unique) C-determining and D-determining subsets are {A,B,D,E} and {A,C,E}, respectively; the corresponding functional consequences, in simplified form, are

$$C = \bar{A}D + B\bar{E} + A\bar{D}E$$

$$D = \bar{A}C + A\bar{C}E .$$

Circuit Design Based on Functional Deduction

An n-input, k-output combinational circuit is typically specified by a system of equations of the form

$$\begin{aligned} z_1 &= f_1(x_1,\ldots,x_n) \\ &\vdots \\ z_k &= f_k(x_1,\ldots,x_n) \\ 0 &= f_{k+1}(x_1,\ldots,x_n). \end{aligned} \tag{39}$$

The latter equation represents any "don't-care" conditions that may exist on allowable input-combinations.

It was observed as early as 1951 [3] that a system of the form

$$\begin{aligned} z_1 &= g_1(x_1,\ldots,x_n) \\ z_2 &= g_2(x_1,\ldots,x_n,z_1) \\ z_3 &= g_3(x_1,\ldots,x_n,z_1,z_2) \\ &\vdots \\ z_k &= g_k(x_1,\ldots,x_n,z_1,\ldots,z_{k-1}) \end{aligned} \tag{40}$$

may meet the functional specifications of (39) at reduced logical cost. Outputs, that is, may be used to assist in the generation of other outputs. The recursive structure of (40) guarantees that the resulting circuit is free of closed loops. There are cases, e.g., the end-around carry in a one's-complement adder, in which closed loops may be employed with good effect in combinational design [4,5,6]. Such loops, however, introduce the possibility of oscillations and other problems inherent in the design of asynchronous sequential circuits; we therefore confine our attention to loop-free specifications of the form (40). We call the corresponding realizations <u>recursive circuits</u>.

The logical cost of a recursive circuit depends on which outputs are allowed to depend on which other outputs; the sequence $(1,2,\ldots,k)$ specified by (40) is only one of $k!$ possible sequences. No method has hitherto been known for determining a promising sequence prior to working out the actual functions corresponding to that sequence. Recursive circuits have consequently been regarded as difficult to design, even though their potential economy has been well-recognized; the design of such circuits is stated in [7] to be "practicable for synthesizing a net which has not more than two or three outputs."

Functional deduction provides a way to overcome the foregoing difficulties, enabling recursive circuits to be designed conveniently. The following procedure is based on minimizing the number of arguments upon which the output-functions depend.

Step 1. Reduce the original specification (39) to a single equation of the form $\phi(x_1,\ldots,x_n,z_1,\ldots,z_k) = 1$.

Step 2. Calculate the z_i-determining subsets $(i = 1,2,\ldots,k)$.

Step 3. Select a sequence $S_{i_1}, S_{i_2},\ldots,S_{i_k}$ of subsets of $\{x_1,\ldots,x_n, z_1,\ldots,z_k\}$ having the following properties:

(a) S_r is a z_r-determining subset $(r = 1,\ldots,k)$.

(b) S_{i_1} is a subset of $\{x_1,\ldots,x_n\}$;
S_{i_2} is a subset of $\{x_1,\ldots,x_n,z_{i_1}\}$;
S_{i_3} is a subset of $\{x_1,\ldots,x_n,z_{i_1},z_{i_2}\}$; etc.

(c) The subsets $S_{i_1}, S_{i_2},\ldots,S_{i_k}$ are as small as possible.

Step 4. Construct simplified consequences of the form $z_r = g_r$ $(r = 1,\ldots,k)$, where the arguments of g_r are those appearing in S_r.

Example 9. A multiple-output circuit is specified by the equations

$$z_1 = \bar{a} + bc$$
$$z_2 = a\bar{b} + \bar{c}$$
$$z_3 = \bar{a} + \bar{b} + \bar{c}.$$

Let us apply the procedure given on the previous page, with the object of reducing the logical cost of the foregoing specifications.

Step 1: $\emptyset = \bar{a}\bar{c}z_1z_2z_3 + \bar{a}cz_1\bar{z}_2z_3 + a\bar{b}\bar{z}_1z_2z_3 + ab\bar{c}\bar{z}_1z_2z_3 + abcz_1\bar{z}_2\bar{z}_3$

Step 2: Calculation of z_i-determining subsets:

$$\emptyset_{z_1} = \bar{a}\bar{c}z_2z_3 + \bar{a}c\bar{z}_2z_3 + abc\bar{z}_2\bar{z}_3$$

$$\emptyset_{\bar{z}_1} = a\bar{b}z_2z_3 + ab\bar{c}z_2z_3$$

$$a_{z_1} = (a)(a)(a + z_2)(a + c + z_2)(b + z_2 + z_3)(c + z_2 + z_3)$$
$$= abc + az_2 + az_3.$$

Similarly,

$$a_{z_2} = abc + cz_1 + acz_3$$

$$a_{z_3} = abc + az_1 + az_2.$$

Step 3: Two subset-sequences are promising:

Sequence # 1	Sequence #2
$S_1 = \{a,b,c\}$	$S_3 = \{a,b,c\}$
$S_2 = \{c,z_1\}$	$S_1 = \{a,z_3\}$
$S_3 = \{a,z_1\}$ or $\{a,z_2\}$	$S_2 = \{c,z_1\}$

Step 4: Simplified functional consequences:

<u>Sequence #1</u>	<u>Sequence #2</u>
$z_1 = \bar{a} + bc$	$z_3 = \bar{a} + \bar{b} + \bar{c}$
$z_2 = \bar{c} + \bar{z}_1$	$z_1 = \bar{a} + \bar{z}_3$
$z_3 = \bar{a} + \bar{z}_1$	$z_2 = \bar{c} + \bar{z}_1$

Either of the foregoing realizations is more economical than a direct realization of the original specifications. Each requires a single IC package, sequence #1 a quad 2-input NAND and sequence #2 a triple 3-input NAND.

<u>Example 10</u>. The input-logic for a clocked D-latch is defined by the equations

$$U = \overline{C} + \overline{D} \tag{41a}$$

$$V = \overline{C} + D, \tag{41b}$$

where U and V are excitation-signals for a NAND-latch, C is a clock-input, and D is a data-input. A circuit implementing (41a) requires a single NAND-gate; however, (41b) requires an inverter in addition to a NAND-gate. To simplify (41b), we resort to functional deduction. The system (41) is equivalent to the single equation $\phi = 1$, where

$$BCF(\phi) = \overline{C}UV + C\overline{D}U\overline{V} + CD\overline{U}V. \tag{42}$$

We deduce from (42) that C, U, and V are functionally deducible from $\phi = 1$. The corresponding determining subsets are represented by the functions a_C, a_U, and a_V:

$$a_C = UV$$

$$a_U = CD + CV$$

$$a_V = CD + CU.$$

The function a_V implies that $\{C,U\}$ is a V-determining subset. The corresponding functional consequence is specified by (35) as follows:

$$D_D(\phi_V) \leq V \leq \overline{D_D(\phi_{\overline{V}})} ,$$

i.e.,

$$\overline{C}U + C\overline{U} \leq V \leq \overline{C} + \overline{U} .$$

A simplified functional consequence specifying V, therefore, is

$$V = \overline{C} + \overline{U}.$$

A circuit implementing the latter relation requires only a single NAND-gate.

Example 11. Let us suppose that we are to design an asynchronous sequential circuit having inputs a_1 and a_0 and output z. The output is to have the value 1 if and only if the present value of the binary number a_1a_0 is greater than the preceding value. We assume that the signals a_1 and a_0 cannot change simultaneously.

By standard processes of asynchronous-circuit design we arrive at the specification

$$y = \text{maj}(a_1,a_0,y)$$
$$z = \text{maj}(a_1,a_0,\bar{y}),$$

where y is an internal state-variable and where the "majority" function maj is defined by $\text{maj}(x,y,z) = xy + xz + yz$. The foregoing specifications are best implemented by full adders (FA's), which generate majority-functions at their carry-outputs; the resulting circuit is shown in Figure 1.

The circuit of Figure 1 requires two packages, a dual full adder and a hex inverter. Only one-sixth of the inverter-package is employed, however, and the upper full adder provides a sum-output,

$$s = a_1 \oplus a_0 \oplus y,$$

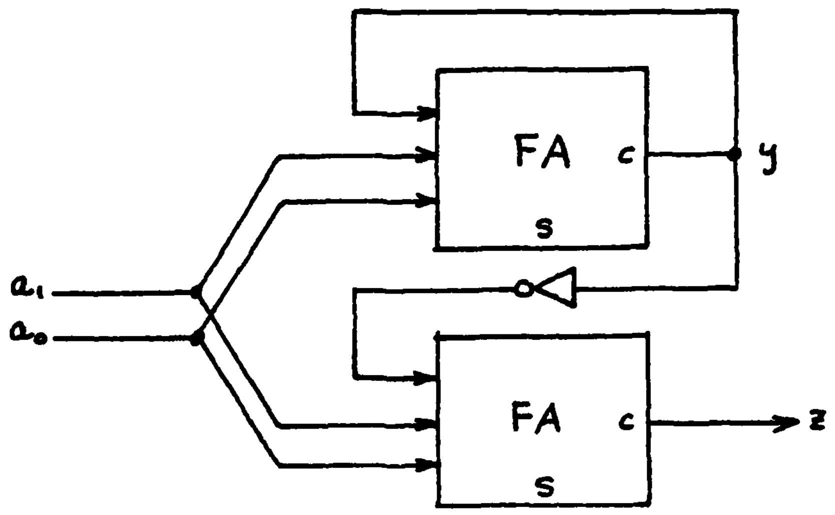


Fig. 1. Asynchronous circuit--original design.

which is not employed at all. These observations lead us to apply functional deduction to the expanded system

$$y = \mathrm{maj}(a_1, a_0, y)$$
$$z = \mathrm{maj}(a_1, a_0, \bar{y})$$
$$s = a_1 \oplus a_0 \oplus y.$$

The foregoing specifications are equivalent to the single equation $\phi = 1$, where

$$\phi = \bar{a}_1\bar{a}_0\bar{s}\bar{y}\bar{z} + (\bar{a}_1a_0 + a_1\bar{a}_0)(s\bar{y}z + \bar{s}y\bar{z}) + a_1a_0syz.$$

Thus

$$\phi_z = \bar{a}_1a_0s\bar{y} + a_1\bar{a}_0s\bar{y} + a_1a_0sy$$
$$\phi_{\bar{z}} = \bar{a}_1\bar{a}_0\bar{s}\bar{y} + \bar{a}_1a_0\bar{s}y + a_1\bar{a}_0\bar{s}y,$$

whence

$$a_z = (a_0 + s)(s + y)(a_1 + a_0 + s + y)(a_1 + s)(a_1 + a_0 + s + y)$$
$$(s + y)(a_1 + a_0 + s + y)(a_1 + s)(a_0 + s),$$

i.e.,

$$a_z = a_1a_0y + s.$$

A result (surprising to this investigator) of the function a_z is that one of the z-determining subsets is $\{s\}$. The corresponding z-consequence is specified by the interval

$$D_{\{a_1,a_0,y\}}(\phi_z) \leq z \leq \overline{D_{\{a_1,a_0,y\}}(\phi_{\bar{z}})} \ ,$$

i.e.,

$$s \leq z \leq \bar{\bar{s}} \ .$$

Thus, z is given by

$$z = s \ .$$

The corresponding circuit, shown in Figure 2, requires only one-half of a dual full-adder package.

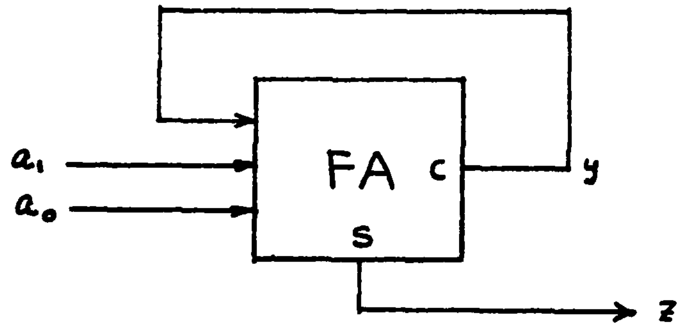


Fig. 2. Asynchronous circuit--modified design.

References

1. Brown, F.M., "Inferential Processor," Final Report, AFOSR/SCEEE Summer Faculty Research Program, August 1980.

2. Boole, G., An Investigation of the Laws of Thought. London: Walton, 1854

3. Harvard University Computation Laboratory, Synthesis of Electronic Computing and Control Circuits. Cambridge: Harvard Univ. Press, 1951.

4. Kautz, W.H., "The necessity of closed loops in minimal combinational circuits, IEEE Trans. Comput., v. C-19, pp. 162-164, February 1970.

5. McCaw, C.R., "Loops in directed combinational switching circuits," Stanford Electronics Laboratories, Tech. Rep't. no. 6208-1, April 1963.

6. Short, R.A., "A theory of relations between sequential and combinational realizations of switching functions," S.E.L. Rep't. no. 098-1, Dec. 1960.

7. Kobrinskii, N.E. and B.A. Trakhtenbrot, Introduction to the Theory of Finite Automata. Amsterdam: North-Holland Publ. Co., Sect. VI.3, 1965.

PART B

INFERENTIAL ANALYSIS OF RELATIONAL DATABASES

Donald Keith Taylor

FOREWORD

Database-processing is an important potential application of the proposed inferential processor. We show in this study that propositional deduction may be used to determine the functional dependencies in a relational database, from which (as is well-known) the keys for the database may be determined. In particular, we have developed and programmed a two-part algorithm: the first part generates the functional dependencies of the relation; the second part uses these dependencies, together with rules for propositional inference, to generate the keys of the relation. The algorithm is programmed in the logical language PROLOG.

TABLE OF CONTENTS

TABLE OF CONTENTS (CONTINUED)

LIST OF ILLUSTRATIONS

CHAPTER I

INTRODUCTION

A major problem in the design and use of computers is that of storing, retrieving, and updating large quantities of non-numerical data. This problem is usually managed by storing these data in a database. Several types of databases exist; however, the relational database has the simplest and most regular structure. This structure makes the relational database attractive for use in large, high-speed data retrieval systems employing associative memories and parallel processors.

The relational model is based on the idea that a database containing information about a particular object (e.g., a university class-schedule) can be viewed as a relation on a set of attributes; the attributes for a class-schedule would be the course number, the room number, the professor's name, and so on. The data of the relational database are stored in a simple tabular form, one row for each record, and one column for each attribute.

The data in each row of the table are accessed by using a key of the database. A key in a relational database is a subset of its attributes which "unlocks" the database: if the value of each attribute in a key is specified, a unique row of the table can be specified. The keys of a relational database are sometimes very difficult to locate; however, examination of the functional dependencies inherent in a database will generate the desired keys.

The functional dependencies of a database have many uses in modern database theory. However, no clearly defined generation method for these dependencies has been developed. Using the recently proven fact that propositional (Boolean) logic can be used to characterize the functional dependencies inherent in a relational database, an algorithmic procedure to generate these dependencies will be derived. By applying Boolean analysis to these dependencies, an algorithm will be developed to determine the keys of relational database. The two preceding algorithms will be joined together to form the FD-Key algorithm. The FD-Key algorithm has the capability to generate the functional dependencies of a relational database; using these dependencies, the keys of the database may be located. To demonstrate the feasibility of the FD-Key algorithm, the logic programming language, PROLOG, will be used to generate the functional dependencies and keys of a given relational database.

In Chapter II, the basic concepts of relational databases are discussed, emphasizing the terms, components, and properties of such databases. Also, some associated problems of utilizing relational databases are explored.

Chapter III presents some fundamental rules and properties of propositional logic and Boolean analysis. Also, the equivalence of propositional logic and relational databases is discussed. An algorithm to generate the keys of a database from its functional dependencies is developed. This algorithm is later used as one of the main components of the FD-Key algorithm.

In Chapter IV, the algorithm to generate the functional dependencies from a relational database is developed. As an example, the functional dependencies and keys for a given relation are derived.

Chapter V discusses the basic concepts of the logic programming language PROLOG. Using these concepts, Chapter VI presents the syntax rules and system requirements for correct implementation of the FD-Key algorithm developed in Chapters III and IV.

Suggestions for future work involving the FD-Key algorithm are presented in Chapter VII. Chapter VIII contains a brief summary of the work and conclusions presented in this thesis. The flowcharts of the FD-Key algorithm presented in this thesis are contained in Appendix A. Appendix B is made up of the actual PROLOG software used to execute the FD-Key algorithms. Finally, Appendix C contains executions of the FD-Key algorithm in PROLOG for several sample relations.

CHAPTER II

INTRODUCTION TO DATABASES

Database Models

A typical database is organized into three different parts: a collection of interrelated data, the hardware necessary to store the data, and the software required to use the data in a real-world application. The database must accurately represent some undertaking in the real world, and it must be at the user's disposal. The currently available hardware imposes a structure upon the data. This structure is called a schema, and it defines the data model used in creating the database. Each model is given a name which represents the way data are viewed by the users. The three currently used structures are the network, hierarchy, and relational models. The database systems that are curently in existence were proposed and studied in many different reports by several authors [1,2,3,4,5,9,12,14,15,17,18,19,21,23].

The network model was first proposed by the Committee on Data System Language, (CODASYL). This model consists of various blocks of data organized in a network. The access time for some blocks of data is very fast, but the user must set up the structure of the system, which cannot be altered once the data have been stored.

The second data structure is the hierarchical data model. Here, data blocks with similar characteristics are accessed by a common data path. Hence, access time between data blocks with similar information is very small, but access time between blocks with very dissimilar data can be very large.

The third data structure is the relational model developed by E. F. Codd [10]. In a relational database, the data are normalized into a form where the relationships among data items appear in a two-dimensional tabular form. Most users have very little trouble in understanding this data model since the two-dimensional table is a familiar method of conveying information. This thesis will use the relational data model exclusively.

Relational Databases

The previous discussion presented some general concepts of data models, but to fully understand relational databases, the accepted conventions, properties, and formal definitions of a relational database must be explained. Henceforth, the use of the word "database" will refer to a relational database.

In a database, the two-dimensional table is called a relation. The columns of the relation are labeled with unique names called attributes, and the rows are called tuples. The data values in the relation are chosen from several sets of values called domains. Each attribute has a domain and several attributes may share the same domain. For example, if a relation has two attributes, say part number and serial number, the attributes are different, but their domains could be the same set of numbers. A more formal definition of a relation is now given, since some of the basic terminology has been discussed.

Definition. Given a set of domains D_1, D_2,..., and D_n, R is a relation on these n sets if it is a collection of n-tuples, $<d_1,d_2,\ldots,d_n>$, such that d_1 is an element of D_1,...,and d_n is an element of D_n.

The usual method of representing attributes in a relation is to allow letters near the beginning of the alphabet to stand for individual attributes, and letters near the end of the alphabet to stand for sets of attributes. For example, C could represent the attribute COURSE in Fig. 1, and X could represent the set of attributes {NAME, COURSE, TIME, ROOM NUMBER}. The union of two sets of attributes, X and Y, is denoted by the concatenation XY, and ABC represents the set of attributes {A,B,C}. The relation R on the set of Attributes X in Fig. 1 is written as R(X). If X is broken into two sets, Y={COURSE, TIME} and Z={NAME, ROOM NUMBER} where X=YZ, then R(X) is the same as R(Y,Z).

NAME	COURSE	TIME	ROOM NUMBER
Green	Psychology	8:00	112
Green	Psychology	10:00	112
Stewart	Chemistry	2:00	106
Stewart	Chemistry	8:00	104
Jones	Mathematics	12:00	210
Smith	Psychology	9:00	104
Johnson	Physics	9:00	210

Fig. 1. Relation R(X).

__Database dependencies__. In a database, several relationships exist among the attributes. One of the main relationships is that of functional dependency (FD). Before dependencies can be discussed, the representation of a data value in a tuple must be explained. Let r be a tuple in the relation R(X) on the set of attributes X, where the set of attributes Y is contained in X. The tuple of values of r for the set of attributes Y is denoted by r[Y].

__DEFINITION__. Given a relation R, two sets of attributes X and Y, the __functional dependency__, X -> Y, holds in R, (or relation R satisfies X -> Y), if and only if for any two tuples v and w in R, v[X] = w[X] implies v[Y] = w[Y].

__DEFINITION__. A dependency s is a __consequence__ of a set of dependencies S if for all relations R, s holds in R if all the dependencies of S hold in R.

Functional dependencies are used extensively in designing relations that are free from data storage and retrieval errors. These errors are called __insertion__, __deletion__, and __rewriting anomalies__. The insertion anomaly is the use of undefined or null values in the table of a relation. The removal of a part of a tuple, causing the loss of other information, is called a deletion anomaly. The rewriting anomaly can easily be explained by the following example. Suppose the functional dependency A->B holds in the relation R(X), and there exist tuples $t_1 = <a_1,b_1,c_1>$ and $t_2 = <a_1,b_1,c_2>$ in R(X). Then if t_1 is changed to $<a_1,b_2,c_1>$, the tuple t_2 must rewritten as $<a_1,b_2,c_2>$. If t_2 is not changed, an anomaly will exist in the relation since the dependency A->B will no longer hold.

__Database schemata and keys__. A __relation schema__ is a description of a single relation consisting of the relation name, a set of attributes, and a set of dependencies. The state (instance or extension) of a relation schema is simply a table of data that conforms to the set of dependencies and uses the attributes contained in the relation schema. A database schema, D, is the set of relation schemata in the database. The state of a database, D, is a mapping of relation states to the schemata of D.

The concept of a set of key attributes (or simply a key) existing in a database is vital to the retrieval of information stored in a database. Once a key has been located, any other information stored in the database can be accessed.

DEFINITION. A subset Y of X is a key for R(X) if and only if Y->X and there is no proper subset Z of Y such that Z->X.

In other words, a key of a relational database is a subset of its attributes that "unlocks" the information stored in the database: if the data values for a key are specified, a unique row of the table is identified.

A notion of a superkey is closely related to the notion of key. A superkey is a set of attributes containing a key as a subset . Consider the relation R(X) shown in Fig. 2, on the set of attributes X = {A,B,C,D}. The set Z = {A,D} is a key of R(X); thus one of the superkeys of R(X) is the set Y = {A,B,D}.

A	B	C	D
a_1	b_1	c_1	d_1
a_2	b_2	c_2	d_1
a_1	b_1	c_1	d_2
a_2	b_3	c_1	d_2

Fig. 2. Relation R(X).

An important but difficult task to be completed before a database can be used is that of determining the set of keys for a given relation. To solve this problem, the set of functional dependencies must either be known or found from the relation. A procedure to generate these dependencies and the keys for a relation is presented later in this thesis.

CHAPTER III

PROPOSITIONAL LOGIC AND THE EQUIVALENCE THEOREM

As discussed in Chapter II, the determination of a set of keys for a relation in a database can be a difficult task. However, once a key has been located, the data stored in the database can be easily accessed. It would be very desirable, therefore, to have a method of key generation for a relation. The aim of the following discussion is to present a method to locate the keys of a relation in a database using the functional dependencies of the relation. In later chapters, this algorithm will be used as a major part of the FD-Key generation algorithm. The method of locating the keys will be developed by examining the equivalence between propositional logic and database dependencies. Before this equivalence can be discussed, some basic ideas of propositional logic and Boolean analysis will be presented.

Propositional Logic

Propositional logic deals with statements that are assigned a truth value. Each statement is called a *proposition*, and it can have only one truth value, either true or false.

These statements are denoted by propositional variables A,B,C,.... Using the logic operations & (AND) and => (IMPLY), an implication $A_1 \& A_2 \& \ldots \& A_n => B_1 \& \ldots \& B_k$ can be created. This implication is said to be true if and only if all of the B_j's are true or at least one of the A_i's is false. Hence, this implication can be viewed as a statement (proposition). Normally, & is represented by simple juxtaposition of the variables. For example, the above implication may also be written as $A_1 A_2 \ldots A_n => B_1 \ldots B_k$. It should be noted that in this thesis the symbol (=>) is used for conditional implication. *Normally, this symbol is used for* logical implication, and the symbol (->) is used for conditional implication. However, the symbol (->) is reserved in this thesis for use with functional dependencies; to avoid notational confusion, therefore, the symbol (=>) is used for conditional implication. The following discussion presents some basic ideas of propositional logic [8].

A fundamental inference-rule of propositional logic is that of hypothetical syllogism. This rule states that the conclusion below follows from its premises.

Major Premise: X=>Y.

Minor Premise: Y=>Z.

Conclusion: X=>Z.

The proposition X=>Z is said to be a logical consequence of the set of propositions {X=>Y,Y=>Z}. In general, we have the following:

Definition. The proposition F is a logical consequence of a set of propositions S, if for every truth assignment P, the proposition F is true under P when all the propositions of S are true under P.

In propositional logic, deduction, (the generation of a conclusion from a set of premises), is performed by invoking various inference rules. These rules state that a specific conclusion can be obtained from a specific set of premises. While these rules work and are useful, a more simplified method of deriving conclusions would be very useful.

Boolean Analysis

Propositions satisfy a set of mathematical laws that are used to define a Boolean algebra. The relation => (conditional implication) of propositions can be translated into the relation $\leq$ (inclusion) of Boolean algebra. In particular, the statement

If X is true, then Y is true

can be represented by the two equivalent expressions

$X=>Y$, and

$X \leq Y$.

The information in these expressions can also be presented in two types of equations. These equations can either be in the "equals-zero" or "equals-one" form of Boolean algebra. The equals-one form is found by complementing the left side of the arrow and forming the logic OR of this result with the right side. For example, the equals-one form of the proposition $X=>Y$ is given by $X' + Y = 1$. The equals-zero form is found by complementing the right side of the arrow and forming the logic AND of this result with the left side of the arrow. For the previous example, the equals-zero form would be $XY'=0$. The equals-one form states that "X is false or Y is true" is a true statement. The equals-zero form states that "X is true and Y is false" is a false statement. Hence, the propositions $X=>Y$, $Y=>Z$, and $X=>Z$ can be represented as Boolean equations $XY'=0$, $YZ'=0$, and $XZ'=0$, respectively. It is a property of Boolean algebra that a sum is equal to zero if and only if each of its summands is equal to zero; hence, the above equations can be written as one equation, i.e.,

$XY'+YZ'+XZ'=0$. Each of the above summands is made up of variables. A single variable, either complemented or uncomplemented, will be called a _literal_, and the summands in the above equation will be called _terms_. Each term consists of a single literal or a product of literals in which no literal appears more than once. A term p is included in a term q if all of the literals of q are contained in p. An _SOP_ (sum of products) formula is a single term or a sum of terms. Two important types of terms will now be defined.

Definition. An _implicant_ of a function F is a term p such that p is included in F.

Definition. A _prime implicant_ of a function F is an implicant p of F such that, for any term q, if p is included in q and q is included in F then p and q are equal.

Blake canonical form. In 1937, A. Blake [6] showed that the sum of all prime implicants of a Boolean function G is a canonical form for that function. We shall call this the _Blake canonical form_ for G and denote it by BCF(G).

There are several methods of generating the Blake canonical form of a Boolean function. This thesis will only deal, however, with the method of _iterated consensus_, which is based upon the following definitions.

__Definition__. Two terms p and q are said to have a literal in opposition if

(i) term p contains a variable A that is uncomplemented, and

(ii) term q contains the complemented variable A'.

__Definition__. Let two terms T_1 and T_2 of a Boolean formula F have exactly one literal in opposition, i.e., let $T_1 = X'P$ and $T_2 = XQ$, where P and Q are terms such that PQ is not equal to zero. Then the __consensus__ of T_1 and T_2 is formed from the product PQ by

(i) deleting the two opposing literals and

(ii) deleting any repetitions of a literal.

The method of generating the BCF of a Boolean function using iterated consensus is given below.

__Definition__. Given a Boolean formula F, BCF(F) can be generated by the following procedure.

(i) Express F as an SOP formula.

(ii) Persist in the following operations as long as possible:

(a) Append to the formula the consensus of two of its terms, unless the consensus is included in a term already present.

(b) Delete any term that is included in another term.

__Definition__. An SOP formula G is said to be __formally included__ in an SOP formula F if every term of G is included in some term of F.

The following two theorems will be presented without proofs. For a more formal presentation, see Blake [6].

__Theorem__. An equation F=0 is a conclusion of the equation G=0 if and only if the function F is included in the function G.

__Theorem__. Let F and G be SOP formulas. Then F is included in G if and only if F is formally included in BCF(G).

To clarify this Theorem, let us examine the following expressions (hypothetical syllogism):

	Propositions	Equations
Major Premise:	X=>Y	XY'=0
Minor Premise:	Y=>Z	YZ'=C
Conclusion:	X=>Z	XZ'=0

After forming the SOP formula G = XY' + YZ', BCF(G) can be found by iterated consensus: BCF(G) = XY' + YZ' + XZ'. But, XZ' is formally included in

BCF(G), so XZ' = 0 is a conclusion of the equation G = 0. This conclusion is equivalent to the proposition X=>Z, and hence the same result is found by two different but equivalent methods. As mentioned earlier, a simplified method for inferring conclusions was desired. Using the Blake canonical form to generate a conclusion from a given set of propositional premises, stated as equations, is such a method. For a more detailed study of this procedure see [7].

Equivalence Between Propositional Logic And Databases

For a given set of propositions {A=>B,C=>D} and a corresponding set of functional dependencies {A->B,C->D}, the syntactical similarity of the sets is very apparent. However, this similarity does not necessarily imply that two corresponding elements of these sets are equivalent. Fortunately, Sagiv, et. al. [21] has proved the following theorem. This theorem states that a set S of functional dependencies is equivalent to a corresponding set of propositions S^*, where S^* is obtained by replacing the dependency symbol(->) with the conditional implication symbol (=>).

Equivalence Theorem. Let F be a functional dependency and let S be a set of dependencies. Then the following are equivalent:

(i) The functional dependency F is a consequence of the set S of functional dependencies.

(ii) The proposition F^* is a logical consequence of the set S^* of propositions.

This theorem states that the set S = {A->B,B->D} of functional dependencies has an equivalent set S^* = {A=>B,B=>D} of propositions, which is generated by replacing the symbol (->) with (=>). Further, since the proposition A=>D is a logical consequence of S^*, the equivalent functional dependency A->D is a consequence of S.

This theorem is a very bold statement. It allows any database problem concerning functional dependencies to be solved by the techniques of propositional logic and guarantees the solution to hold for the dependencies of the database. Since the available tools of propositional logic are generally much easier to implement than the inference rules for dependencies, a very difficult database problem may easily be solved with propositional logic. Hence, the preceding method of iterated consensus may be used to generate the solutions for a given problem concerning the dependencies of a database.

Key generation. As an example of the power of this theorem let us examine the relation R(X) in Fig. 1 of Chapter II. The following functional dependencies exist in this relation. Note that the attributes are replaced by one-letter symbols to make the variable manipulations clearer.

NAME->COURSE	N->C
NAME,TIME->ROOM,COURSE	NT->RC
NAME,ROOM->COURSE	NR->C
COURSE,TIME->NAME,ROOM	CT->NR
COURSE,ROOM->NAME	CR->N
TIME,ROOM->COURSE,NAME	TR->CN

After writing the preceding six dependencies in their equivalent propositional logic forms, the following equations are generated by complementing the right sides of the equivalent propositions and forming the logic AND of this result with the left sides of the propositions.

N=>C	C'N=0

NT=>RC	R'NT + C'NT=0
NR=>C	C'NR=0
CT=>NR	CN'T + CR'T=0
CR=>N	CN'R=0
RT=>CN	C'RT + N'RT=0

Since these equations are in equals zero form, they are equivalent to the single equation G = 0, where the function G is the logical sum of their left members, i.e.,

G = C'N + R'NT + C'NT + C'NR + CN'T + CR'T + CN'R + C'RT + N'RT.

To generate the keys associated with a relation, a method based upon the one developed by Delobel and Casey [13] will be used. For a given relation, the minterm M, which is the juxtaposition of all of the attribute symbols, is always a superkey of the relation. If K is the juxtaposition of all the attributes of a key of the relation and if G = 0 is the equation representing the set of dependencies of the relation, then the implication

[G = 0] => [K = M]

defines all of the keys of the relation.

For the above implication to be true, either G = 0 must be false, i.e., $G = 1$, or K must be equal to M, i.e., K and M must both be false, or K and M must both be true. Hence, the above implication can be expressed as the equivalent equation

$$G + KM + K'M' = 1.$$

By applying some generally known properties of propositional logic and Boolean algebra to the above equation, the following equivalent forms can be derived.

$$K(M + G) + K'(M' + G) = 1$$

$$K(M'G') + K'(MG') = 0$$

$$G'M \leq K \leq (G'M')'$$

$$G'M \leq K \leq G + M$$

Let us examine the formula G. Since G represents the original dependencies of the relation, each term of G will contain at least one complemented attribute symbol. Hence, G may not include a minterm containing only uncomplemented attribute symbols. The minterm M containing all of the attribute symbols in uncomplemented form is therefore not included in G. Thus M is included in G', i.e.,

$$M \leq G'.$$

The foregoing inclusion is equivalent to the Boolean equation

$$M = G'M;$$

therefore, the expression

$$G'M \leq K \leq G + M$$

is equivalent to

$$M \leq K \leq G + M,$$

which is equivalent in turn to

$$M \leq K \leq BCF(G + M).$$

The reason that BCF(G + M) is used is that it includes all of the information available in terms that contain the fewest possible attribute symbols.

To determine the keys for a relation, only the terms of BCF(G+M) that contain no complemented attribute-symbols are considered. This can be explained by re-examining the bounds on K. The minterm M is the product of all of the attribute symbols, and it forms the lower bound on K. So M must be included in K, and hence K can only include uncomplemented attribute symbols. But K must be included in BCF(G + M); therefore the terms of BCF(G + M) that are keys must contain only uncomplemented attribute symbols.

From the previous example for Relation R(X) of Fig. 1 where C = COURSE, N = NAME, R = ROOM NUMBER, and T = TIME, the term M is found to be M = CNRT. Using the set {N->C, NT->RC, NR->C, CT->NR, CR->N, TR->CN} of functional dependencies for this relation, together with the equivalence theorem, the set {N=>C, NT=>RC, NR=>C, CT=>NR, CR=>N, TR=>CN} of equivalent propositions is generated. The formula

$$G = N'RT + C'RT + C'NT + NR'T + CN'T + CR'T + CN'R$$

is produced by converting each proposition into an equation of equals-zero form, forming the sum G = 0 of all of these equations, and writing the formula G. Adding the term M to G and calculating BCF(G + M) yields the result

$$BCF(G+M) = CT + RT + NT + C'N + CN'R.$$

Using the expression

$$M \leq K \leq BCF(G + M)$$

for the bounds on the unknown key K, the relation

$$CNRT \leq K \leq CT + RT + NT + C'N + CN'R$$

is generated. Now by examining the terms of BCF(G+M) containing no complemented variables, the keys CT, RT, and NT for the relation R(X) of Fig. 1 are found. Also, the superkeys of a relation can be found by concatenating any number of uncomplemented attribute symbols in the relation to the symbols of a key. Therefore, CNT, CRT, NRT, and CNRT are superkeys of the relation.

If the set of functional dependencies for a relation is known, the above procedure will generate all keys and superkeys that exist in the relation. If an algorithm existed to generate the functional dependencies of a relation, then the generation of keys for a relation could be implemented on a computer or dedicated processor designed to perform propositional analysis. This desired dependency generation algorithm has been developed, and it will be presented in the following chapter. Once both of these algorithms have been presented, they can be combined into a single FD-Key generation algorithm.

CHAPTER IV

GENERATION OF FUNCTIONAL DEPENDENCIES

For the key generation algorithm of Chapter III to be applied, the functional dependencies of a relation must be known. We present in this chapter a procedure for generating the functional dependencies of a relation directly from the rows (tuples) defining that relation. When combined with the key-generation algorithm, this procedure enables the keys of a relational database to be derived quickly and conveniently. We call the combined procedure the *FD-Key Algorithm*.

The FD-Key algorithm may be used to solve a number of problems. Suppose a programmer were assigned the task of setting up a large database; then the generation of the keys could be a very tedious and time consuming task. By using the FD-Key algorithm, the programmer could simply insert the database into the computer system, execute the algorithm, and receive the keys and functional dependencies of the database as outputs.

As another example, suppose a database could be updated by several different users. That is, several users could be changing data values of tuples in the database. This process might create anomalies in the database. Hence, a functional dependency for the original database might no longer hold in the updated version of the database. This type of error may be detected in the following manner. After each change of data. the FD-Key algorithm could be executed on the new database. If the functional dependencies generated from the new database were different from the functional dependencies of the original database, then the recent data changes had violated the integrity of the database. Therefore, the data-updates should be examined for an error.

This chapter will present the section of the FD-Key algorithm that generates the functional dependencies of a relation. To fully understand the operation of this section of the algorithm, the flowcharts of Appendix A and the examples contained in this chapter should be closely examined.

The algorithm makes extensive use of partitions, whose definition we now recall [16].

Definition. A partition P of a non-empty, finite set S is a collection of non-empty subsets of S. The partition is denoted by $P = \{B_1, B_2, \ldots, B_k\}$; the subsets $B_1,\ldots,B_2$ are called the blocks of the partition P. The blocks of a partition must satisfy the following two conditions.

(i) The intersection of any two blocks, B_i and B_j for i not equal to j, is the empty set.

(ii) The union of all the B_i's is the set S.

Generation Algorithm

The problem to be solved can be stated as follows: given a relation R and a specific attribute A contained in the relation, generate the functional dependencies of R that contain A on the right side of the arrow. The desired dependencies will have the form X->A, where X may be the concatenation of several attribute symbols. By continuing this process for all of the attributes, the set of functional dependencies for the given relation will be generated.

The algorithm to generate the functional dependencies manipulates the data in the relations of the database in three different ways. The first data manipulation involves partitioning. Specifically, the tuples of the original relation are placed into a number of relations containing two sub-relations, each of which is generated by using a two-block partition of the data values for an attribute in the original relation. After formation of the sub-relations, the second type of data manipulation is performed. Here, the tuples in each pair of sub-relations are compared and a Boolean sum of attribute symbols is generated. This sum actually represents the key for the two tuples in the sub-relations being examined. By repetitive generation of these sums for each pair of tuples, the keys of the pair of sub-relations can be found. After all of these sums are generated for the pair of sub-relations, a product of sums (POS) formula is generated by forming a Boolean product of all of the sums. Each pair of sub-relations will be subjected to this procedure, and a group of POS formulas will be generated. Each of these formulas will either be zero, (indicating that no key exists for the pair of sub-relations) or a product of sums. If a formula is zero, this means that no functional dependency can be found for the chosen attribute of the original relation. If all of the formulas are products of sums, however, a third type of data manipulation is needed to generate the functional dependencies of the relation.

This final data manipulation involves some techniques of Boolean algebra. All of the POS formulas generated from the pairs of sub-relations are multiplied together to form one large POS formula. This last POS formula is converted to a sum of products (SOP) formula and simplified as much as possible. The SOP formula now contains all of the information needed to produce the set of functional dependencies for the original relation. Each term of the SOP formula contains the attribute symbols that are on the left sides of the arrows in the functional dependencies that have the chosen attribute from the original relation on the right side of the arrow.

In the development of the functional dependencies for a relation, the following three assumptions will be made.

(1) The data-values of the relation will not be updated during the development period.

(2) The relation will contain a finite number of tuples.

(3) The relation has a finite number of attributes.

The algorithm to generate the functional dependencies of a relation will now be presented in seven steps.

Step 1. Choose an attribute to appear on the right side of the functional dependencies.

Step 2. Generate the set, L, consisting of one entry for every unique data value in the column under the chosen attribute.

Step 3. Generate a sequence ($P_0{}^*, \ldots, P_n{}^*$) of partitions of the set L in the following manner:

(i) Let i=0. $P_i{}^*$ contains all the data values of the set L in one block.

(ii) If the number of elements in the largest block of $P_i{}^*$ is less than or equal to two, stop the operations. Otherwise do (iii).

(iii) Let i=i+1, and generate a new $P_i{}^*$ that contains 2^i blocks. The blocks in the new partition are foundby splitting each of the blocks in the preceding partition into two disjoint blocks whose cardinality differs at most by one. If a block in the preceding partition contains an odd number of elements, the left block of the new pair of blocks will contain one more element than the new right block. Go to (ii) and repeat.

Step 4. Generate another sequence of partitions ($P_1, \ldots, P_k$), of two blocks each, in the following manner:

(i) Let i = 1, and j = 1. Set $P_i = P_j{}^*$.

(ii) P_{i+1} is made up of two blocks such that the left block contains the left half of each block of P_j^*. The right block of P_{i+1} contains all the elements in P_j^* not present in the left block of P_{i+1}. If a block of P_j^* contains an odd number of elements, the extra element is placed in the left block of P_{i+1}.

(iii) If P_j^* is the last partition, stop the procedure. Otherwise, let $i = i+1$, and $j = j+1$.

The previous two steps have been designed to provide maximum skewing of the partitions. That is, the number of elements in the left block of each P_i is as large as possible. This procedure will minimize the number of data comparisons necessary in the sixth step of the total algorithm.

Step 5. Generate n copies of the relation being tested, where n is the number of generated P_i partitions. Split each copy of the relation into two sub-relations according to the data values found in the blocks of the P_i's. Delete the columns corresponding to the attribute being tested from each of these copies. Let R_{i1} and R_{i2} denote the ith pair of sub-relations, where the subscripts i1 and i2 represent the tuples associated with the data values contained in blocks one and two, respectively, of partition P_i. The generation of the sub-relations is outlined below.

(i) Let i=1, j = the numerical position of the chosen attribute in the list of attribute symbols, P_i = the ith two-block partition of the set of data values for the chosen attribute, and T1 = the first tuple of the relation.

(ii) If the jth data item of T1 is in the left block of P_i, then place T1 in the sub-relation R_{i1}. Otherwise, place T1 in the sub-relation R_{i2}.

(iii) If T1 is the last tuple of the original relation, then go to (iv). Otherwise, let T1 = the next tuple of the relation, and go to (ii).

(iv) If i=n, stop this procedure. Otherwise, let i=i+1, let T1 = the first tuple of the relation and go to (ii).

Step 6. Generate a Boolean formula EI for each sub-relation R_i. The formula generated will be in a product of sums, POS, form. That is, a sum of literals logically multiplied by other sums of literals. These formulas are generated by the following procedure.

(i) Let i=1, obtain both the first tuple t_1 of the sub-relation R_{i1} and the first tuple t_2 of R_{i2}. If R_i doesn't exist, go to Step 7.

(ii) If $t_1=t_2$, abort the dependency algorithm since no functional dependencies exist with the chosen attribute on the right side of the arrow. Otherwise go to (iii).

(iii) Compare the data values under corresponding attributes of each tuple. If any pair of data values are distinct, insert their attribute name into a sum S. Continue this procedure until all pairs of data-values in t_1 and t_2 have been exhausted, then go to step (iv).

(iv) Insert the sum S as a product in the POS formula EI. Go to step (v).

(v) If t_2 was the last tuple of R_{i2} and t_1 was the last tuple of R_{i1}, let i=i+1 and go to step (i). If t_2 was the last tuple of R_{i2} and t_1 was not the last tuple of R_{i1}, replace t_1 with the next tuple of

R_{i1} and t_2 with the first tuple of R_{i2}; then go to step (ii). If t_2 was not the last tuple of R_{i2}, replace t_2 with the next tuple of R_{i2} and go to step (ii).

It was stated previously that maximum skewing of the partitions will minimize the number of tuple comparisons. For example, assume the relation R_i contains n+k tuples such that the sub-relations R_{i1} and R_{i2} contain n and k tuples, respectively. If k is much smaller than n, so that $k=n-p$, where $p>0$, then the number of tuple comparisons required to generate the formula EI is $n \times k = n \times (n-p) = n^2 - np$. But if k and n are equal, then the number of comparisons required is $n \times n = n^2$. And if $k=n+1$, then the number of comparisons would be even larger, i.e., $n \times k = n(n+1) = n^2+n$. So it is clearly evident that maximum skewing of the partitions P_i's is necessary to minimize the number of tuple comparisons.

Step 7. Generate the functional dependencies for the relation. These dependencies will contain the chosen attribute on the right side of the arrow. The procedure is outlined below.

(i) Let the function EA be composed of the product of all the EI functions previously found in the sixth step of the algorithm.

(ii) Convert the POS form of EA to a SOP form by multiplying the products and deleting any terms that contain all the literals of another term in EA.

(iii) Create a dependency with each term T_i of EA and the attribute A, and place each of these dependencies into a set DEP. These dependencies will be of the form T_i->A, and DEP will be equal to {T_1->A,..., T_m->A}. The set DEP now contins all of the dependencies from the original relation that have the chosen attribute on the right side of the arrow.

To clarify the operation of the algorithm, the relation R(X) in Fig. 3 will be used to generate an example for each step of the algorithm. This relation can be found in [21]. Also, some supplementary examples are given to clarify steps of the algorithm that are overly simple when applied to this relation.

A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_2	c_2	d_2
a_2	b_3	c_1	d_1
a_1	b_1	c_1	d_2
a_2	b_3	c_1	d_2
a_1	b_2	c_2	d_1

Fig. 3. Relation R(X).

The first step of the algorithm states that an attribute to appear on the right side of the functional dependencies must be selected. So, in the example relation R(X), the attribute B will be chosen. The second step of the algorithm generates the set L of data values associate with this attribute. For our example, this set is $L=\{b_1,b_2,b_3\}$.

Performing the next step of the algorithm on our relation the sequence of partitions (P_0^*,P_1^*) will be generated. These partitions are sets that contain other sets, hence

$$P_0^* = \{\{b_1b_2b_3\}\}$$

$$P_1^* = \{\{b_1b_2\},\{b_3\}\}$$

is the correct manner of representing these partitions. This notation contains many braces, and it needs to be simplified. Henceforth, all partitions will be denoted by deleting the braces of the blocks of the partitions, e.g.,

$$P_0^* = \{b_1b_2b_3\}$$

$$P_1^* = \{b_1b_2,b_3\}.$$

To further clarify this step of the algorithm, let us examine the flowchart of Fig. 9 and another set of data values, namely, L1 = $\{a_1,a_2,a_3,a_4,a_5,a_6,a_7,a_8,a_9,a_{10},a_{11},a_{12}\}$. The following partitions will be generated by this step of the algorithm.

$$P_0^* = \{a_1a_2a_3a_4a_5a_6a_7a_8a_9a_{10}a_{11}a_{12}\}$$

$$P_1^* = \{a_1a_2a_3a_4a_5a_6,a_7a_8a_9a_{10}a_{11}a_{12}\}$$

$$P_2^* = \{a_1a_2a_3,a_4a_5a_6,a_7a_8a_9,a_{10}a_{11}a_{12}\}$$

$$P_3^* = \{a_1a_2,a_3,a_4a_5,a_6,a_7a_8,a_9,a_{10}a_{11},a_{12}\}$$

Now, performing the fourth step of the algorithm on the sequence of partitions generated from the set L, the partitions

$$P_1 = P_1^* = \{b_1b_2,b_3\}$$

$$P_2 = \{b_1b_3,b_2\}$$

will be generated. Again, a flowchart and another example is given to clarify this step of the algorithm. Examining the flowchart of Fig. 10 and the sequence of partitions for the set L1, the sequence

$$P_1 = P_1^*$$

$$P_2 = \{a_1a_2a_3a_7a_8a_9,a_4a_5a_6a_{10}a_{11}a_{12}\}$$

$$P_3 = \{a_1a_2a_4a_5a_7a_8a_{10}a_{11},a_3a_6a_9a_{12}\}$$

$$P_4 = \{a_1a_3a_4a_6a_7a_9a_{10}a_{12}, a_2a_5a_8a_{11}\}$$

of partitions is produced.

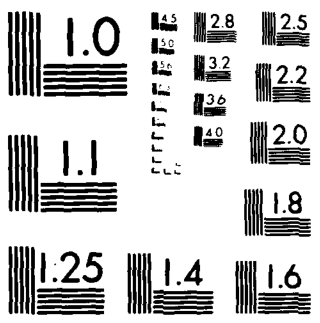
The fifth step of the algorithm will generate several copies of the original relation R(X). Using the partitions P_1 and P_2 of the set L, the copies R_1' and R_2' of Fig. 4 for relation R(X) of Fig. 3 will be generated. This figure shows how the two copies R_1' and R_2' appear before the columns corresponding to attribute B are deleted.

The first copy R_1' of R(X) is partitioned in the following manner. Block one of partition P_1 contains the data values b_1 and b_2. Therefore, any tuple of R(X) containing these data values under the column B will be placed in the sub-relation R_{11}. Since block two of P_1 contains only b_3, the sub-relation R_{12} will only contain tuples that have the data value b_3 under the column B. Similarly for the second copy R_2' of R(X), R_{21} will contain tuples that have the data values b_1 and b_3 under the attribute B. Likewise, R_{22} will only contain tuples that have the value b_2 in the column B.

AD-A111 550 KENTUCKY UNIV LEXINGTON DEPT OF ELECTRICAL ENGINEERING F/G 9/2
INFERENTIAL PROCESSOR.(U)
JAN 82 F M BROWN, D K TAYLOR AFOSR-81-0115
UNCLASSIFIED AFOSR-TR-82-0037 NL

2 of 2
AD A111550

END
DATE FILMED
4-82
DTIC



MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU

	B	A	C	D
R_{11}	b_1	a_1	c_1	d_1
	b_2	a_1	c_2	d_2
	b_1	a_1	c_1	d_2
	b_2	a_1	c_2	d_1
R_{12}	b_3	a_2	c_1	d_1
	b_3	a_2	c_1	d_2

(a) Copy R_1' of relation R(X).

	B	A	C	D
R_{21}	b_1	a_1	c_1	d_1
	b_3	a_2	c_1	d_1
	b_1	a_1	c_1	d_2
	b_3	a_2	c_1	d_2
R_{22}	b_2	a_1	c_2	d_2
	b_2	a_1	c_2	d_1

(b) Copy R_2' of relation R(X).

Fig. 4. Copies of R(X) partitioned according to P_i's.

After the column of data associated with the attribute B is deleted, the new relations R_1 and R_2 are as shown in Fig. 5.

	A	C	D
R_{11}	a_1	c_1	d_1
	a_1	c_2	d_2
	a_1	c_1	d_2
	a_1	c_2	d_1
R_{12}	a_2	c_1	d_1
	a_2	c_1	d_2

(a) Copy R_1 of relation R(X).

	A	C	D
R_{21}	a_1	c_1	d_1
	a_2	c_1	d_1
	a_1	c_1	d_2
	a_2	c_1	d_2
R_{22}	a_1	c_2	d_2
	a_1	c_2	d_1

(b) Copy R_2 of rela'.ion R(X).

Fig. 5. Copies of R(X) with attribute B removed.

The sixth step of the algorithm will be illustrated by examining the two relations of Fig. 5 and the flowchart of Fig. 12. In the generation of E1 for relation R_1, the tuples $t_1=<a_1,c_1,d_1>$ and $t_2=<a_2,c_1,d_1>$ are the first two tuples examined. Now by comparison of data values, it is found that only attribute A will be in the first sum S. At this point, therefore, E1 consists of only one sum, (A).

After performing the tests in step (v), the tuple t_2 will be changed to $t_2=<a_2,c_1,d_2>$, and t_1 will remain the same. These two tuples are compared, and the new sum S=A+D is generated. The new sum S is placed in the formula E1, and this changes E1 to E1=(A)(A+D). Again the tests in step (v) are performed, and this time both t_1 and t_2 will be changed to $t_1=<a_1,c_2,d_2>$ and $t_2=<a_2,c_1,d_1>$. After all the tuples of R_1 have been examined, the formula E1 is found to be

E1=(A)(A+D)(A+C+D)(A+C)(A+D)(A)(A+C)(A+C+D),

which is equivalent to the formula E1=A. Performing the same operations on the relation R_2 will yield the result

E2=(C+D)(C)(C)(C+D)(A+C+D)(A+C)(A+C)(A+C+D),

which is equivalent to E2=C. For the final step of the algorithm, the formula EB and its derivation is shown below.

EB=(E1)(E2)

EB=(A)(C)=AC

Therefore the set DEP contains only th[illegible]ctional dependency AC->B. By repeated application of this algorithm for the other attributes, the complete set

{B->A, AC->B, B->C}

of functional dependencies for relation R(X) of Fig. 3

can be produced.

Recalling the key generation algorithm presented in the preceding chapter, the following results can be found from the set of functional dependencies. Since the relation R(X) contains the attributes A, B, C, and D, the term M will be equal to ABCD, and from the preceding set of dependencies the equivalent equation $G = 0$ can be derived, where $G = A'B + AB'C + BC$. Using the technique of iterated consensus on M and G, the following results can be obtained.

$$BCF(G+M) = A'B + AB'C + BC' + BD + ACD$$

$$ABCD \le K \le A'B + AB'C + BC' + BD + ACD$$

Therefore the set of keys for the relation R(X) of Fig. 3 is {BD,ACD}, and the corresponding set of superkeys is {ABD,BCD,ABCD}. The results generated above may also be generated by visual observation for this relation. A much larger relation may be very hard to analyze visually, but the preceding algorithm will always generate the desired results. The implementation of this algorithm would be very easy in a language designed for logic programming and character string manipulations. Fortunately, the programming language PROLOG has these capabilities and it is very easy to operate from a user's view. This language is presented in the next chapter and the actual implementation of the FD-Key generation algorithm, using PROLOG, will be presented in Chapter VI.

CHAPTER V

INTRODUCTION TO PROLOG

Basic Structure

In the past few years, several logic programming languages have been developed. One of the most powerful of these is PROLOG, a programming language based on predicate calculus; this language was developed at the University of Marseille starting around 1970. A later, interactive version of PROLOG was implemented on the DECsystem-10 in 1977 [22]. This newer version, containing both an intepreter and a compiler, allows the user to easily write clear, readable, and concise programs. The interpreter aids in the quick development and testing of programs, and also allows access to compiled programs. The compiler produces code that executes ten to twenty times faster than the interpreter, but it is advisable to compile only well-tested programs. Any compiled program can easily be provided with an interpretative interface to the programmer. We present a brief summary in this chapter of the features of PROLOG; for a more detailed description, see [20] and [11].

Components of Prolog

Generally, any object in PROLOG can be called a term. A term can either be a constant, a variable, or a compound term. A constant can be any integer between -131072 to 131071 or an atom. The integers can be written in any base from two to ten. An atom can be any sequence of characters, and any possible confusion with other terms should be eliminated by enclosing the sequence in quotes. For example, 'Rabbit', rabbit, [], and = are all atoms.

A variable is distinguished by an initial capital letter or the leading character "_". Whenever a variable is only referenced once, it can be denoted by the single character "_". For example, Rabbit, X, _32, _result, and _ are all variables.

A compound term is formed with a functor of some arity greater than one. The arity of a functor is the number of terms used as arguments. In the term member(X,[H|T]) for example, the functor "member" has an arity of two since X and [H|T] are the two terms used as arguments. The term [H|T] represents a list, where H is the first element and T is the tail or all remaining elements in the list. An atom may be considered as a functor of arity zero.

The names and arities of functors are totally arbitrary. That is, the programmer can introduce as many different arguments for a desired functor as needed. PROLOG contains several built-in functors used to perform basic system operations.

A PROLOG program consists of a set of <u>procedures</u> which contain <u>clauses</u>. These clauses are made up of terms, organized into two basic forms. The propositional logic form of the first type of clause, called a <u>conditional Horn clause</u>, is of the form

$$A<=B_1\&B_2\&B_3$$

where A is called the <u>head</u> of the clause and B is called the <u>body</u> of the clause. This clause is read "A is true if B_1 and B_2 and B_3 are true." A conditional horn clause may also have the form

$$A<=C_1+C_2$$

where this clause is read "A is true if C_1 or C_2 is true." The second type of Horn clause, known as a <u>unit clause</u>, is a true statement such as

A

which is read "A is true."

The PROLOG language requires the head of a clause to be separated from the body by the symbol :-, which represents the word "if" in a logic statement. Also, any clause must end with a period. For example, the three preceding clauses translated into PROLOG would be written as

$A:-B_1,B_2,B_3.$

$A:-C_1;C_2.$

$A.$

These clauses taken together can be viewed as procedure A, where B_1, B_2, B_3, C_1, and C_2 are goals or other procedures to be called by the PROLOG program. The goals in the body of a procedure are separated by the symbols "," or ";" which represent logical conjunction and disjunction, respectively. These goals are procedures that impose conditions upon the head of the clause.

PROLOG also contains provisions for expressing grammar rules. These rules provide an easy method of parsing strings into specific components, and using these components in any manner specified by the program. The typical grammar rule has the form, LHS-->RHS, and it is read as "a possible form for the left hand side is the right hand side." Any PROLOG procedure can be used as a condition on the right side by simply enclosing the procedure in braces, "{}". Grammar rules may seem very confusing when first encountered, but they can be written as ordinary PROLOG clauses. For example, the grammar rule p(X)-->q(X) can be translated into the clause p(X,S1,S):-q(X,S1,S). As an example, the procedure

```
delim-->"+".
delim-->[].
```

is a rule to remove the character "+" from a list of characters. If "+" were not the first character in the list, the original list would be the result.

Prolog Semantics

PROLOG semantics can be presented in two different ways. The procedural semantics describes the sequence of states through which the program passes during an execution, and the declarative semantics allows the program to be broken down into many independent programs or procedures. These smaller procedures are usually clear and easily executed.

The declarative semantics makes no reference to the ordering of clauses or procedures in a goal or program. This type of semantics is used to recursively define the conditions necessary for the head of a clause to be true. That is, the head of a clause is true if all of the terms in the body of the clause are also true, and each term is true if it, in turn, is the head of a clause instance which is true. Also a term in the body of a clause may be a compound term which is the disjunction of two other terms. For example, the clause

A:-B;C.

is true if the compound term B;C is true, and the compound term is true if either B or C is true.

The procedural semantics depends upon the ordering of clauses in a program, and the goals in a clause, for crucial program control information. The execution of the program depends upon this information, and the reordering of a set of goals or clauses may completely change the function of a clause or program. The execution of a goal is performed by searching for the first clause whose head matches the goal. This is done in a top-down fashion. That is, the matching starts at the top of the program and continues until a match is found. If a match is found, the goals in the body of the clause are executed from left to right in the same manner. If no match is found, the system backtracks to the most recent clause, discards any substitutions caused by that clause. and the search for another match of the original clause is continued from this clause down through the rest of the program.

There is one other type of control information available in PROLOG, called the cut symbol. This symbol, "!", is used as a goal in a clause and always succeeds when it is first called. If PROLOG ever backtracks to the cut symbol, the goal that caused the clause containing the cut symbol to be called will always fail. This symbol allows the programmer to force a goal to succeed or fail after it has been partially executed.

Examples of Prolog Programs

Two simple examples of PROLOG programs will be presented. The first example will consist of a program to solve the following logic problem.

> Bob likes logic.
>
> Mary likes logic.
>
> Bob likes anyone who likes logic.
>
> What does Bob like?

The PROLOG program will consist of two unit clauses and one conditional clause involving the predicate "likes":

```
likes(bob,logic).
likes(mary,logic).
likes(X,Y):-likes(Y,logic).
```

After the program has been interpreted by the computer, the input query

```
                likes(bob,X).
```

will yield the results:

```
                X=logic;
                X=bob;
                X=mary
```

Here, the symbol ";" is used to request an alternate answer for the query after one answer has been found.

As another example, consider the problem of concatenating two lists together to form a third list. The procedure could be formulated as follows:

```
                concatenate([],L,L).
                concatenate([X|L],T,[X|K]):-
                        concatenate(L,T,K).
```

The predicate "concatenate" is defined by the program; that is, PROLOG does not know what this procedure means until it receives these statements. However, the symbols [] and [|] are known to the language. The first clause states that the empty list concatenated with a second list is simply the second list. The second clause states that the list [X|L] concatenated with the list T is the list [X|K] if the list L concatenated with the list T is the list K. When the query

```
concatenate([a,b,c],[d,e,f],K)
```

is presented to this program, the variable K will be returned as the list [a,b,c,d,e,f]. The above procedures are but two of many possible examples, and it should be noted that an excellent source [11] of programming examples exists.

The next chapter contains a description of the operation of the FD-Key algorithm presented in the preceding chapters. This description contains the syntax rules that must be obeyed for proper operation of the program. and some possible modifications that the user may wish to use.

CHAPTER VI

SYSTEM REQUIREMENTS AND SYNTAX

The purpose of this chapter is to present the syntax rules and system requirements for the correct implementation of the FD-KEY algorithm as it is currently programmed using the language PROLOG.

System requirements. For the FD-Key program in Appendix C to run correctly, the user's computing system should meet the following requirements. First, Version 3 of Dec-10 PROLOG or its equivalent must be used. Otherwise, several clauses in the program will not function correctly. For example, any clause that uses the built-in predicate 'read' to input information from a data file will usually have a test for the end of file marker, 'end_of_file'. If an earlier version of PROLOG is used, this marker may be ':-end', and the program will never cease to input information from the data file. Therefore, a loop will be created, and the program will either fail or yield an erroneous result.

The other requirement is that the user's system must have an adequate amount of memory storage available. This is because the FD-Key program generates several files for data input and output during execution. Also, the program and its compiled version require several blocks of storage. It should be noted that the program deletes all of the data files created during execution. The only exception to this is the file 'propa', which contains the functional dependencies of the input relation.

FD-Key algorithm. The FD-Key algorithm consists of two main routines. The first routine is the functional dependency algorithm, presented in Chapter IV. The second routine is the key generation algorithm presented in Chapter III. The operation of these algorithms is explained in the following section of this chapter.

The FD-Key algorithm is designed to perform the following tasks:

(i) Input a list of attribute symbols for a given relation.

(ii) Output the functional dependencies for the relation.

(iii) Output the keys for the relation.

(iv) Output run-times for various routines in the algorithm.

Since the language used is PROLOG, all constants must be in lower case letters. For example, the list of attribute symbols for relation R(X) of Fig. 3, would be [a,b,c,d]. If the attribute symbols in the list were capital letters, PROLOG would interpret the contents of the list as unknown variables. This interpretation could lead to meaningless answers or to the failure of the entire program. It should be noted that the ordering of the attribute symbols is very important. The symbols must be in the same order as the columns of the relation. For example, if the list for R(X) of Fig. 3 were changed to [b,a,d,c], the algorithm would not generate the correct functional dependencies. The generated keys would be invalid for the relation.

As the FD-Key algorithm is currently programmed, the relation to be examined must be stored in a particular form. Each tuple of the relation must be stored as a list. For example, the tuple

$$<a_1,b_1,c_1,d_1>$$

would be stored as the list

$$[a_1,b_1,c_1,d_1].$$

Each list must be followed by a period or the program will not be able to input the data correctly. The file '^base' contains the lists that correspond to the tuples of tne given relation. To run the main algorithm, the system must be in PROLOG, and the compiled version of the FD-Key program must be restored. For example, consider the output of Appendix C for the relation R(X) of Fig. 3. The program is called by the predicate

```
mainthing([a,b,c,d]).
```

The argument, [a,b,c,d], of the predicate is simply the list of attribute symbols for the relation R(X). When this predicate is executed, the relation in the file 'dbase' is examined; the functional dependencies for this relation are stored in the file 'propa', and the keys of the database, along with the run-times for various routines in the algorithm, are output to the user.

If the user wishes to generate the keys for a relation whose functional dependencies are known, the set of dependencies must be stored in the file 'propa'. Each dependency must be written as a logical proposition followed by a period. Thus, the functional dependency A->B would be stored in the file 'propa' as

```
a=>b.
```

To call the key generation section of the program, the following predicate is used:

```
solve_for_keys([a,b,c,d]).
```

In this example, the argument [a,b,c,d] is the list of attributes for the relation containing the functional dependencies found in the file 'propa'.

To make any other predicate of the program available to the user, a public statement must be used to declare the predicate, and the new program must be compiled. For example, to declare the predicate

```
concatenate(X,Y,Z).
```

of Chapter V, the statement

```
:-public concatenate/3.
```

must be inserted in the program. The format of this statement is

```
:-public name/arity.
```

In this statement, the name of the predicate is separated from the number of its arguments by the slash.

The flowcharts of the FD-Key algorithm can be found in Appendix A. These flowcharts are written at a level which will enable the user to translate the algorithm to another language if PROLOG is not available. The PROLOG program of the FD-Key algorithm is found in Appendix B. This program contains numerous comments designed to explain each set of clauses. Generally, the purpose of the set of clauses and a brief example is contained in each comment. Appendix C is made up of some sample runs of the FD-Key algorithm for various relations. Each run contains the input commands to the computer, a listing of the input relation, a listing of the keys for the relation, and a listing of the functional dependencies generated by the algorithm.

CHAPTER VII

PROPOSALS FOR FUTURE WORK

There are a number of ways in which the present FD-Key routine might be improved, i.e., made more efficient or extended in application. Three such improvements are detailed below.

Employment of an inferential processor. Since the key generation routines depend upon two separate calculations of the Blake Canonical Form, a processor capable of performing this calculation in hardware would be very advantageous. Fortunately, an inferential processor has been proposed [8] that can generate the Blake Canonical Form very quickly. This device receives a sum of terms formula from a host computer and outputs the Blake Canonical Form of this sum to the host. By using this type of device, a large portion of the key generation program could be replaced. A revised algorithm to generate the keys is presented below.

(i) Input the set S of functional dependencies for the relation.

(ii) Express the set S as a Sum of Products formula F.

(iii) Add the term which corresponds to the list of attributes to F.

(iv) Output F to the hardware processor.

(v) Input BCF(F) from the processor.

(vi) The set of keys K corresponds to the terms of BCF(F) that contain only uncomplemented literals.

This key generation algorithm clearly reduces the amount of software used, and consequently the cost of processing and the computer time required would be reduced. By using this technique, the algorithm may be speeded up to be used in a real-time data processing situation. Another method of speeding up the key generation routine is presented in the next section.

Multi-valued dependency generation. As the FD-Key algorithm is presently formulated, only functional dependencies of a relation are manipulated. The inclusion of the information contained in the multi-valued dependencies of the relation would speed up the key generation routine. However, two problems remain to be solved before this improvement can be made. First, an algorithm to generate the multi-valued dependencies from the relation would have to be developed. Secondly, an algorithm to translate these dependencies into a sum of products form would have to be created. After these problems are overcome, the new SOP formula could be added to the SOP formula for the functional dependencies, and the resulting Blake Canonical Form of this formula may provide some additional information not contained in the original Blake Canonical Form for the functional dependencies.

Program modifications. Another area of improvement would be to change the PROLOG implementation of the FD-Key algorithm. The new PROLOG program would be different in two ways. First, the routines that manipulate the Boolean formulas would be changed. These new routines would work directly on the sum of products or product of sums formulas instead of a list of lists. This modification would not only speed up the data manipulations, but it would also remove the routine used to parse a formula into a list of lists. Although this modification would involve major revisions in the program routines, a large saving of run-time should be realized.

A second way to improve the efficiency of the program would be to keep as much data as possible in fast memory, that is, not to store data in files for later use in the program. By keeping the data in fast memory, the data retrieval time will be very short and the algorithm would execute more efficiently. This modification would also require major revisions in many of the procedures, but the run-times should be quicker.

Normal form generation. A final area of future work might be the development of an algorithm to produce normal forms of a relation. If an algorithm to generate both functional and multi-valued dependencies existed, a method to generate the normal forms of a relation based upon this algorithm could be produced. The development of this normalization routine should be a very straight-forward, since the normal forms of a relation are generated by examining the keys and dependencies that are associated with that relation.

CHAPTER VIII

CONCLUSIONS

The purpose of this thesis was to present an algorithm capable of generating the keys and functional dependencies of a relational database. The feasibility of this algorithm was demonstrated by implementing it with the computer language PROLOG.

The key generation algorithm is based on a theorem, due to Sagiv, concerning the equivalence of logical propositions and functional dependencies. This theorem allows the formidable problem of key generation to be solved by techniques of Boolean analysis.

An algorithm based on partitioning was then developed to generate the functional dependencies of a database. These two algorithms were combined to form the FD-Key algorithm, which was implemented using the logic-programming language PROLOG.

This implementation involved many different uses of propositional logic, Boolean analysis, and the Blake canonical form for the actual generation of functional dependencies and keys for a relational database. The execution of the FD-Key algorithm imposes a distinct set of computer system requirements. These requirements were presented and some actual executions of the FD-Key algorithm were given as examples.

The FD-Key algorithm provides a convenient method for generating the keys and functional dependencies of a database. This algorithm has produced a solution to a difficult and complex problem of relational databases, namely the identification of the keys necessary to access the information stored in the database, by using the techniques of propositional logic and Boolean analysis.

APPENDIX A

FLOWCHARTS FOR THE FD-KEY ALGORITHM

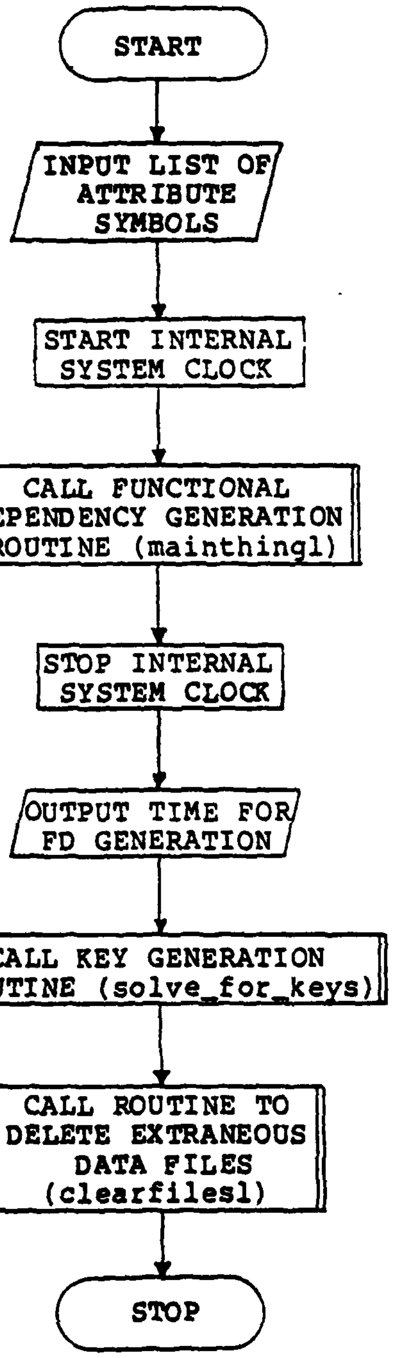
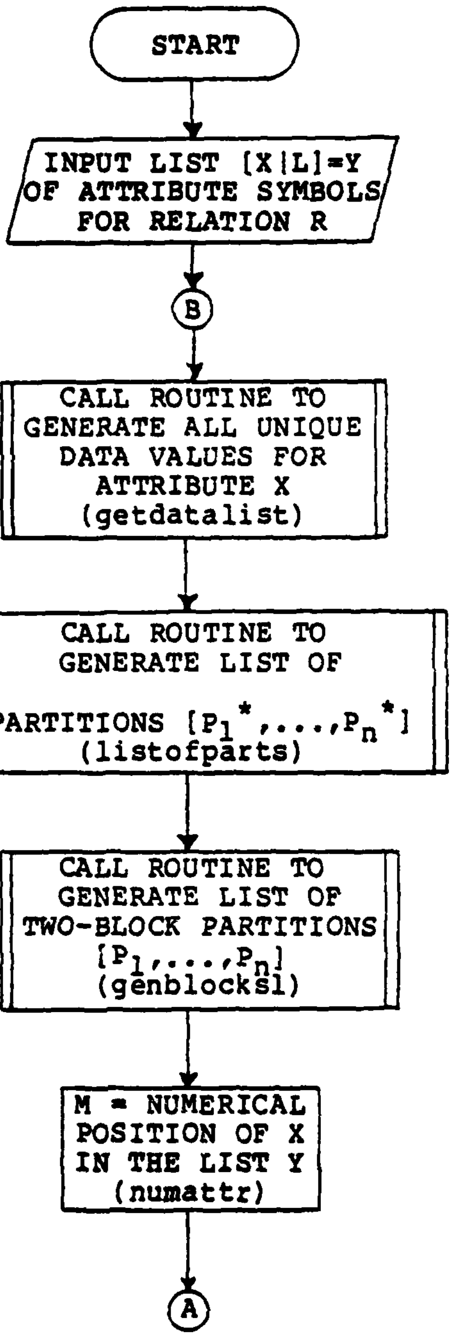
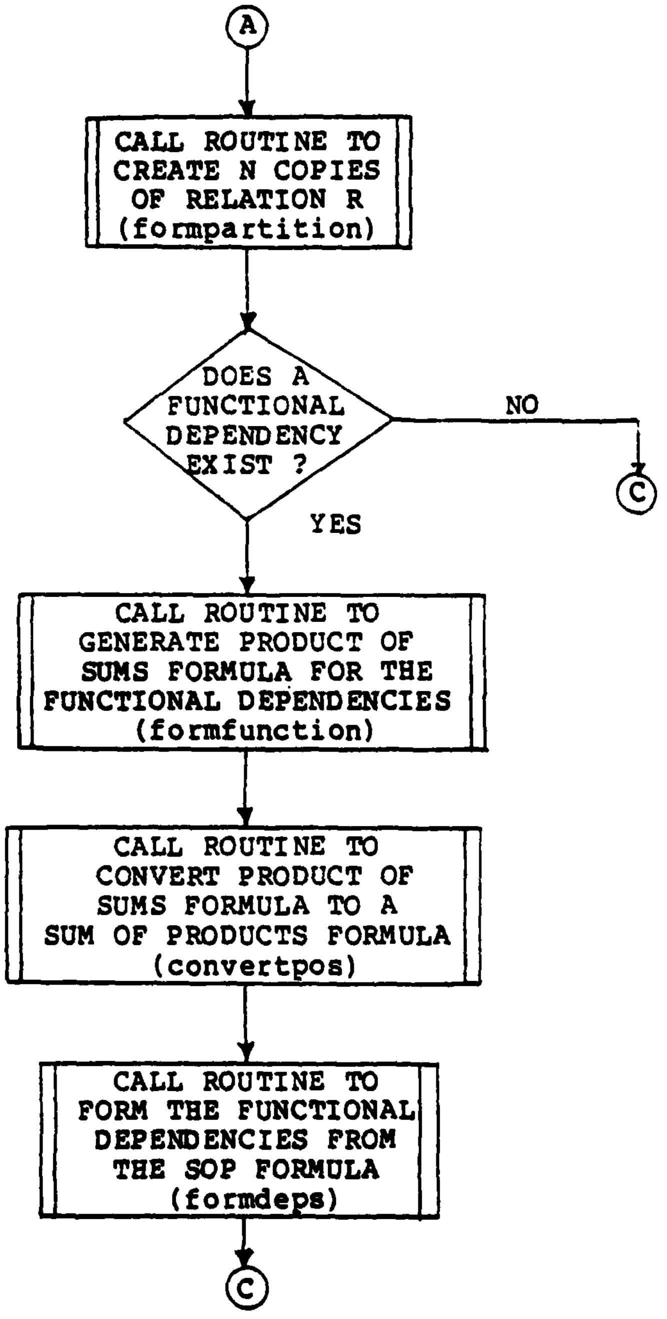

START
INPUT LIST OF ATTRIBUTE SYMBOLS
START INTERNAL SYSTEM CLOCK
CALL FUNCTIONAL DEPENDENCY GENERATION ROUTINE (mainthing1)
STOP INTERNAL SYSTEM CLOCK
OUTPUT TIME FOR FD GENERATION
CALL KEY GENERATION ROUTINE (solve_for_keys)
CALL ROUTINE TO DELETE EXTRANEOUS DATA FILES (clearfiles1)
STOP

Fig. 6 Main algorithm (mainthing).

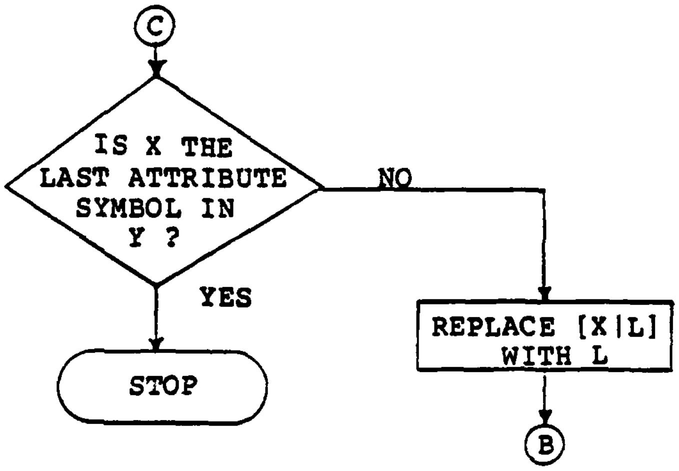

START
INPUT LIST [X|L]=Y OF ATTRIBUTE SYMBOLS FOR RELATION R
B
CALL ROUTINE TO GENERATE ALL UNIQUE DATA VALUES FOR ATTRIBUTE X (getdatalist)
CALL ROUTINE TO GENERATE LIST OF PARTITIONS [P1*,...,Pn*] (listofparts)
CALL ROUTINE TO GENERATE LIST OF TWO-BLOCK PARTITIONS [P1,...,Pn] (genblocks1)
M = NUMERICAL POSITION OF X IN THE LIST Y (numattr)
A

7a.

Fig. 7. Functional dependency routine (mainthing).


A
CALL ROUTINE TO CREATE N COPIES OF RELATION R (formpartition)
DOES A FUNCTIONAL DEPENDENCY EXIST ?
NO
C
YES
CALL ROUTINE TO GENERATE PRODUCT OF SUMS FORMULA FOR THE FUNCTIONAL DEPENDENCIES (formfunction)
CALL ROUTINE TO CONVERT PRODUCT OF SUMS FORMULA TO A SUM OF PRODUCTS FORMULA (convertpos)
CALL ROUTINE TO FORM THE FUNCTIONAL DEPENDENCIES FROM THE SOP FORMULA (formdeps)
C

7b.


C
IS X THE LAST ATTRIBUTE SYMBOL IN Y ?
NO
YES
STOP
REPLACE [X|L] WITH L
B

7c.

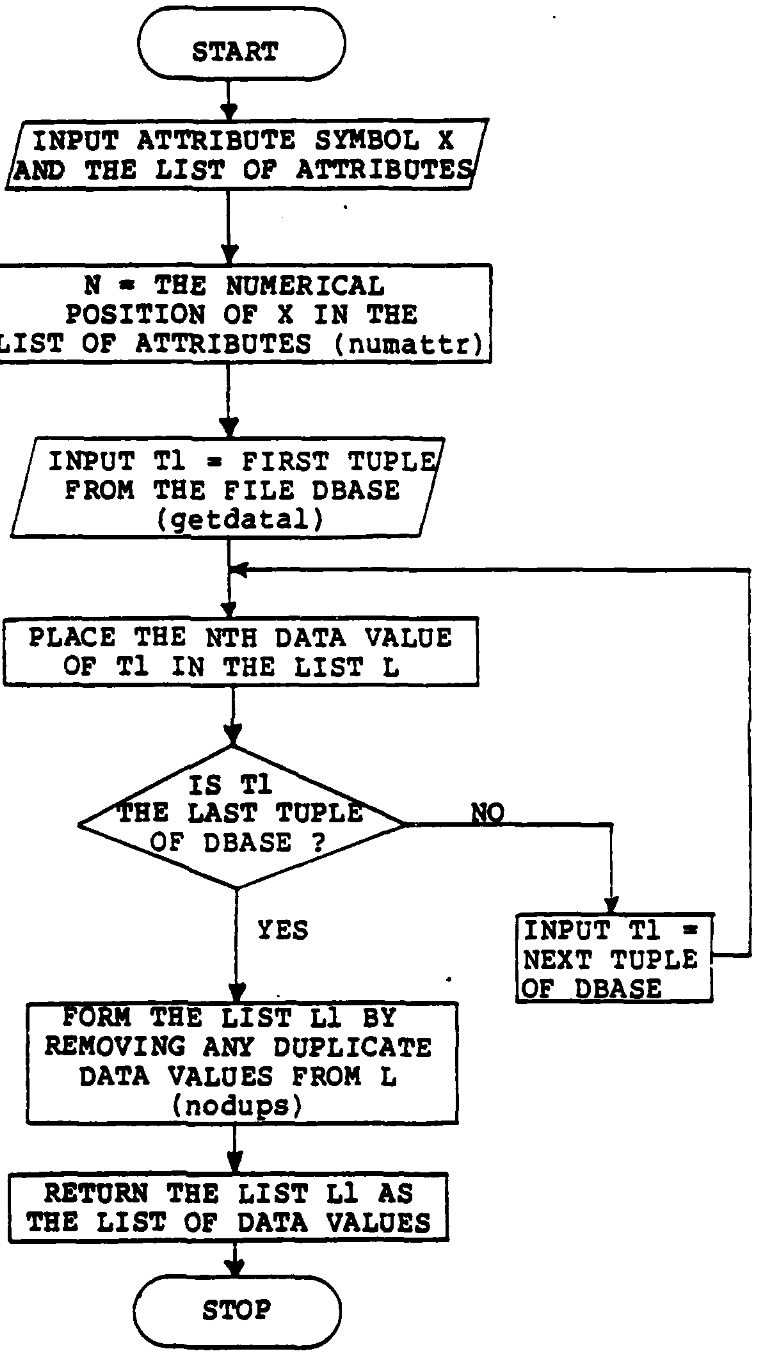

START
INPUT ATTRIBUTE SYMBOL X AND THE LIST OF ATTRIBUTES
N = THE NUMERICAL POSITION OF X IN THE LIST OF ATTRIBUTES (numattr)
INPUT T1 = FIRST TUPLE FROM THE FILE DBASE (getdata1)
PLACE THE NTH DATA VALUE OF T1 IN THE LIST L
IS T1 THE LAST TUPLE OF DBASE ?
NO
INPUT T1 = NEXT TUPLE OF DBASE
YES
FORM THE LIST L1 BY REMOVING ANY DUPLICATE DATA VALUES FROM L (nodups)
RETURN THE LIST L1 AS THE LIST OF DATA VALUES
STOP

Fig. 8. Routine to generate list of unique data items (getdatalist).

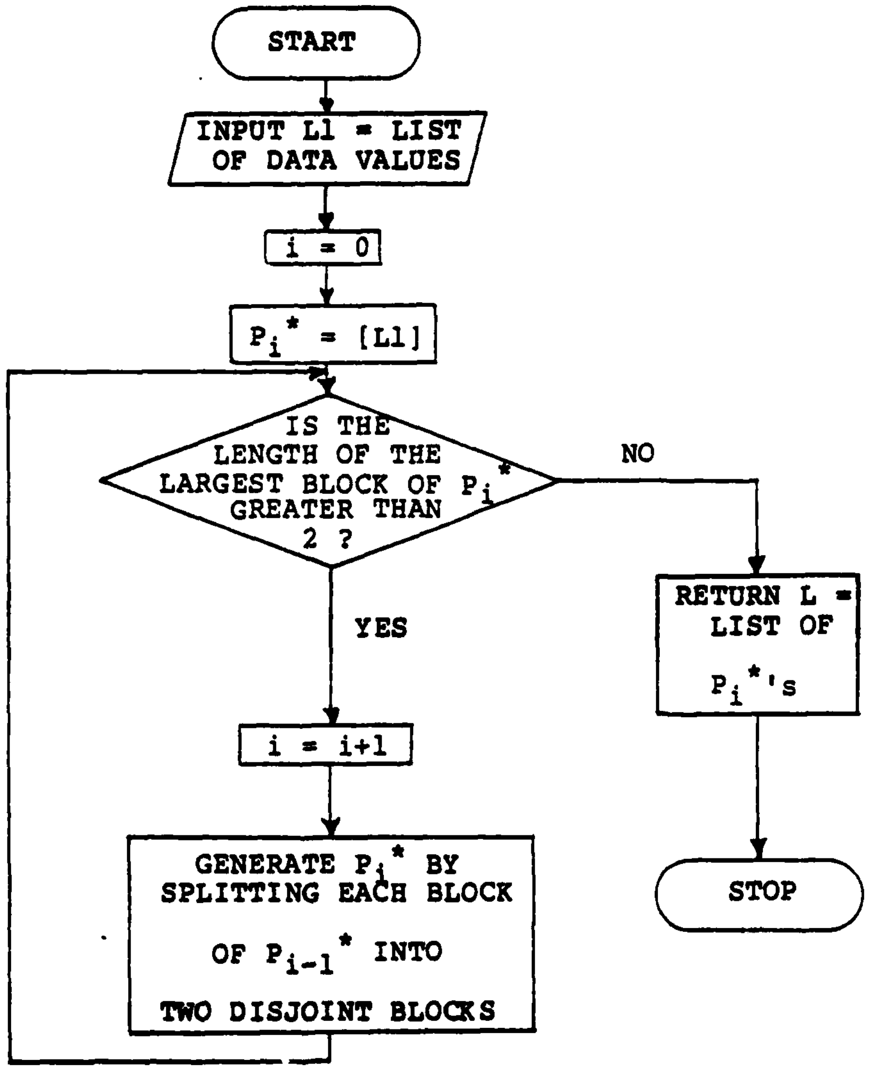

START
INPUT L1 = LIST OF DATA VALUES
i = 0
P_i^* = [L1]
IS THE LENGTH OF THE LARGEST BLOCK OF P_i^* GREATER THAN 2 ?
NO
YES
RETURN L = LIST OF P_i^*'s
i = i+1
GENERATE P_i^* BY SPLITTING EACH BLOCK OF P_{i-1}^* INTO TWO DISJOINT BLOCKS
STOP

Fig. 9. Routine to generate the list of P_i^* partitions (listofparts).

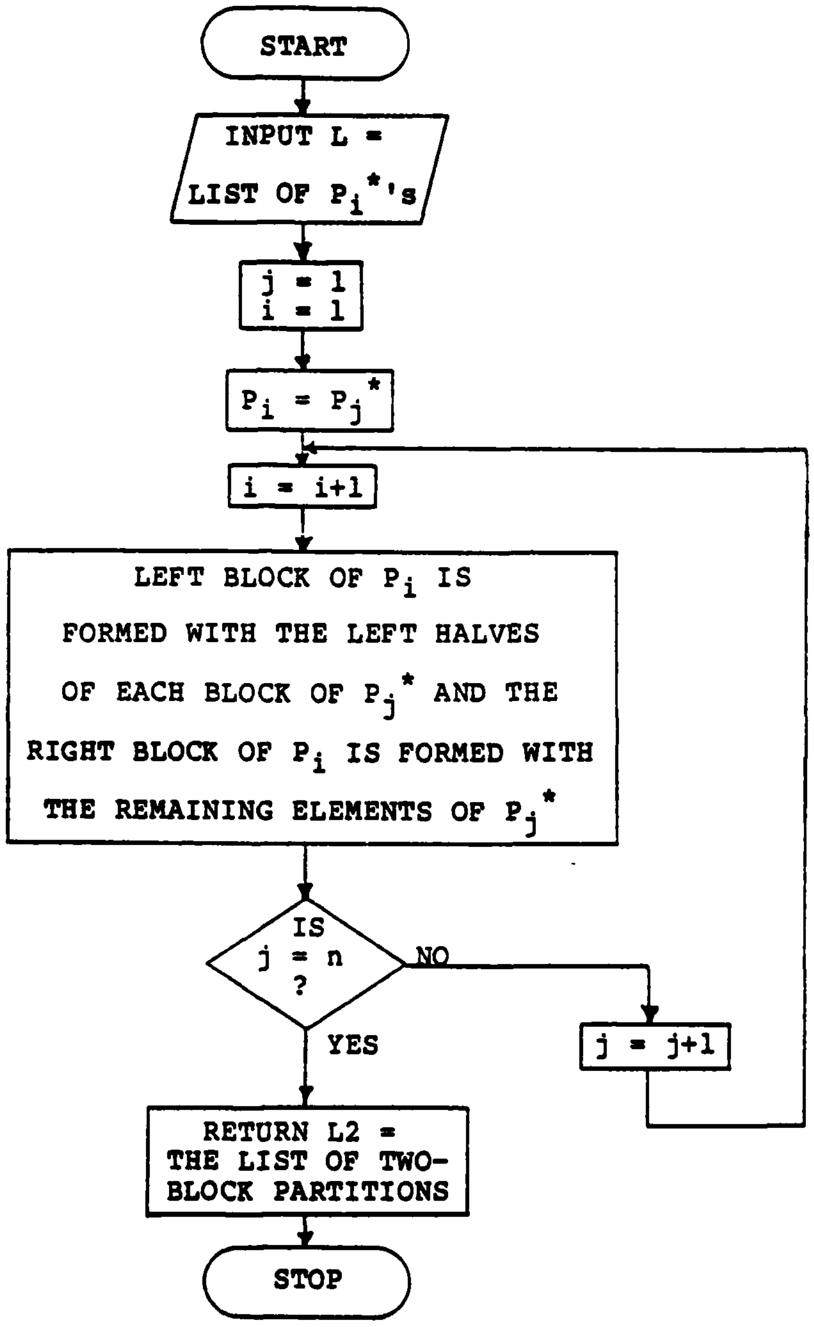
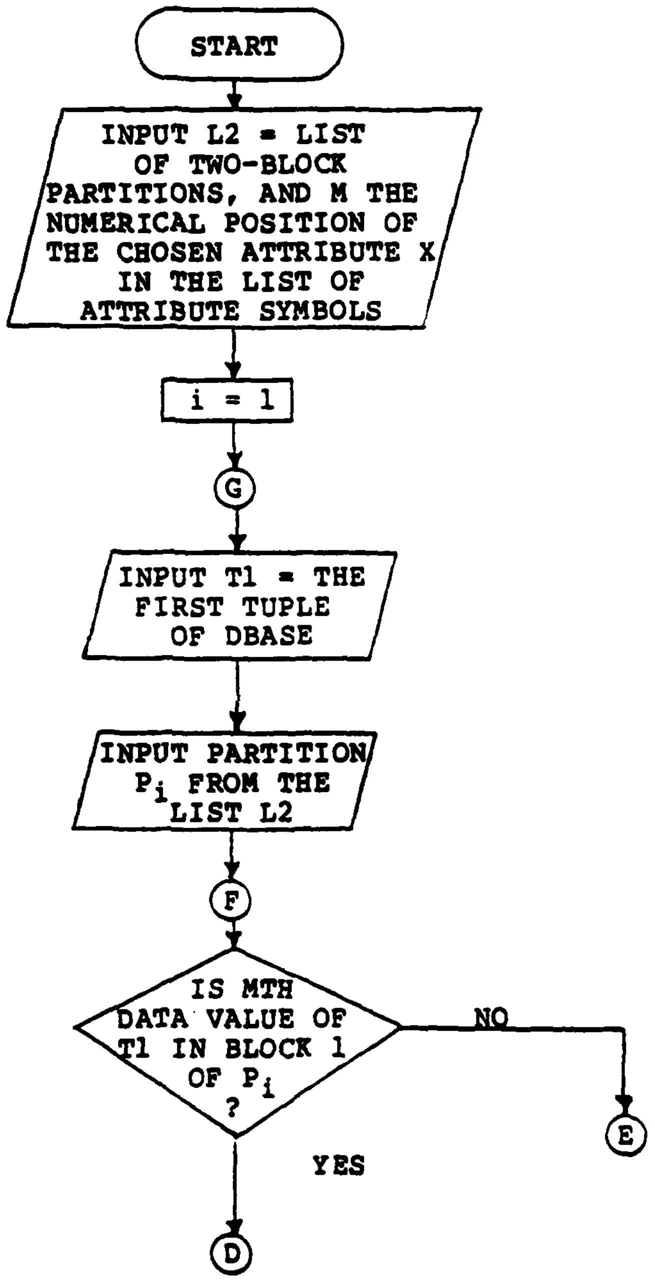
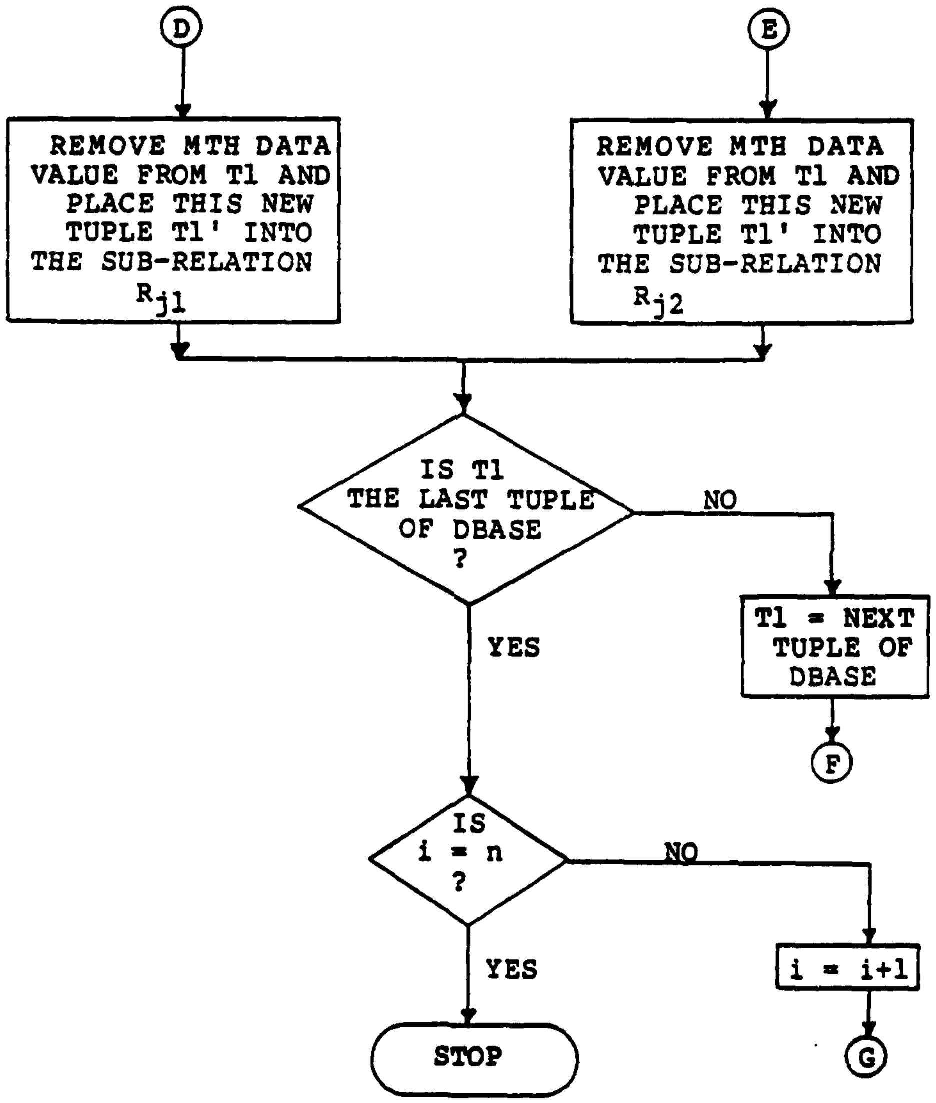

START
INPUT L =
LIST OF P_i^*'s
j = 1
i = 1
$P_i = P_j^*$
i = i+1
LEFT BLOCK OF P_i IS FORMED WITH THE LEFT HALVES OF EACH BLOCK OF P_j^* AND THE RIGHT BLOCK OF P_i IS FORMED WITH THE REMAINING ELEMENTS OF P_j^*
IS j = n ?
NO
YES
j = j+1
RETURN L2 = THE LIST OF TWO-BLOCK PARTITIONS
STOP

Fig. 10. Routine To Generate The List Of Two-block partitions (genblocksl).

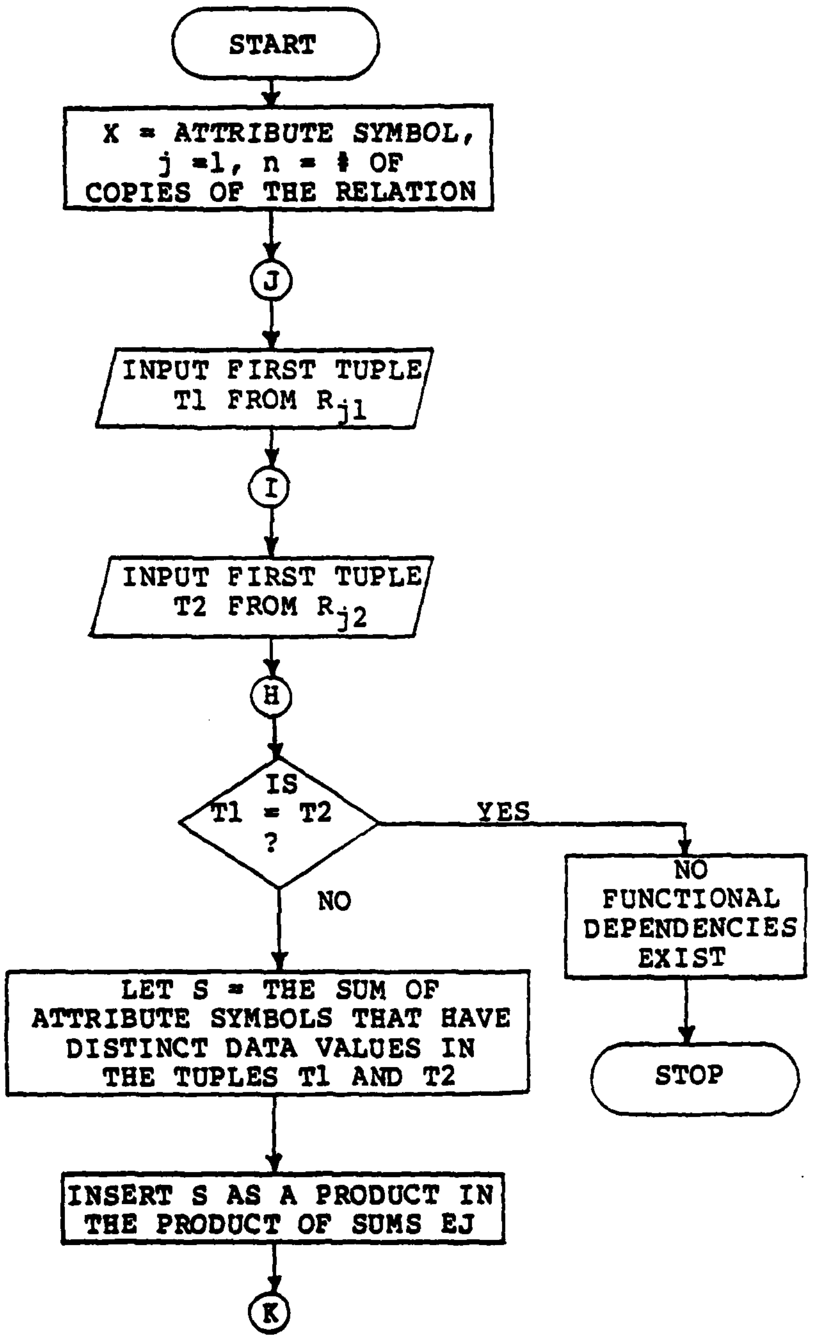

START
INPUT L2 = LIST OF TWO-BLOCK PARTITIONS, AND M THE NUMERICAL POSITION OF THE CHOSEN ATTRIBUTE X IN THE LIST OF ATTRIBUTE SYMBOLS
i = 1
G
INPUT T1 = THE FIRST TUPLE OF DBASE
INPUT PARTITION P_i FROM THE LIST L2
F
IS MTH DATA VALUE OF T1 IN BLOCK 1 OF P_i ?
NO
E
YES
D

11a.

Fig. 11. Routine To Create n Copies Of The Original Relation R (formpartitions).

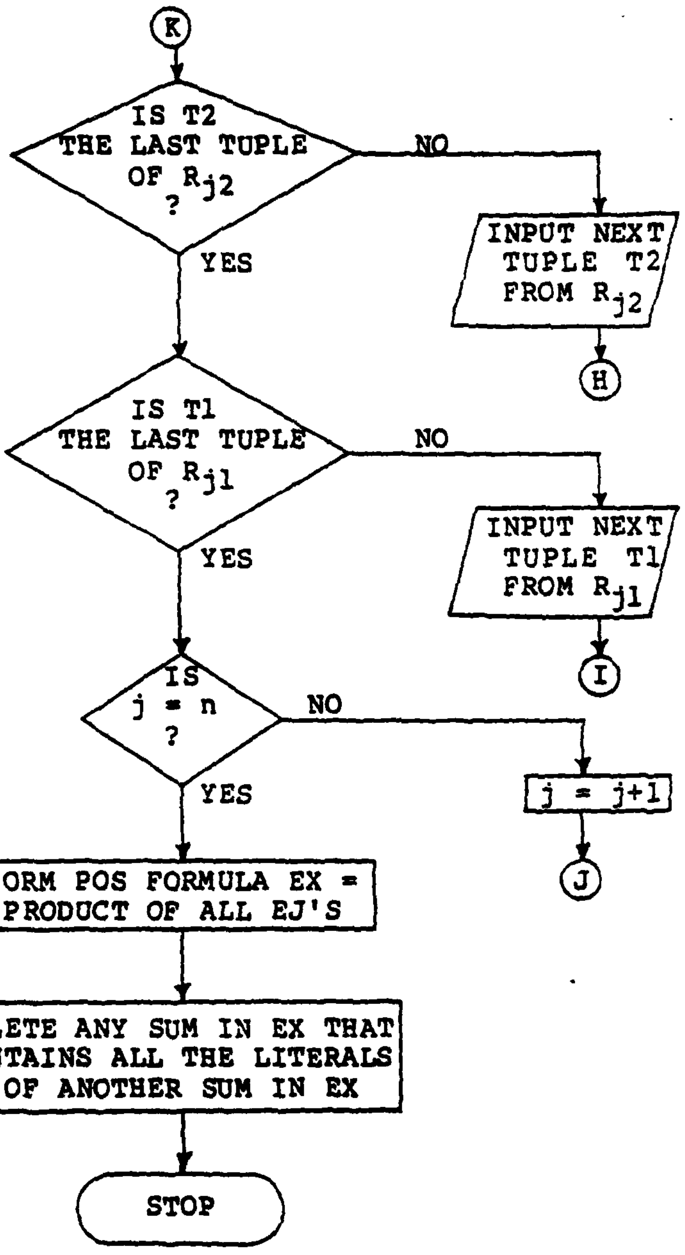

D
E
REMOVE MTH DATA VALUE FROM T1 AND PLACE THIS NEW TUPLE T1' INTO THE SUB-RELATION R_{j1}
REMOVE MTH DATA VALUE FROM T1 AND PLACE THIS NEW TUPLE T1' INTO THE SUB-RELATION R_{j2}
IS T1 THE LAST TUPLE OF DBASE ?
NO
T1 = NEXT TUPLE OF DBASE
F
YES
IS i = n ?
NO
i = i+1
G
YES
STOP

11b.


START
X = ATTRIBUTE SYMBOL, j =1, n = # OF COPIES OF THE RELATION
J
INPUT FIRST TUPLE T1 FROM R$_{j1}$
I
INPUT FIRST TUPLE T2 FROM R$_{j2}$
H
IS T1 = T2 ?
YES
NO
NO FUNCTIONAL DEPENDENCIES EXIST
STOP
LET S = THE SUM OF ATTRIBUTE SYMBOLS THAT HAVE DISTINCT DATA VALUES IN THE TUPLES T1 AND T2
INSERT S AS A PRODUCT IN THE PRODUCT OF SUMS EJ
K

12a.

Fig. 12. Routine to test for functional dependencies and generate the POS formula EJ (formfunction).


K
IS T2 THE LAST TUPLE OF R_{j2} ?
NO
INPUT NEXT TUPLE T2 FROM R_{j2}
H
YES
IS T1 THE LAST TUPLE OF R_{j1} ?
NO
INPUT NEXT TUPLE T1 FROM R_{j1}
I
YES
IS j = n ?
NO
j = j+1
J
YES
FORM POS FORMULA EX = PRODUCT OF ALL EJ'S
DELETE ANY SUM IN EX THAT CONTAINS ALL THE LITERALS OF ANOTHER SUM IN EX
STOP

12b.

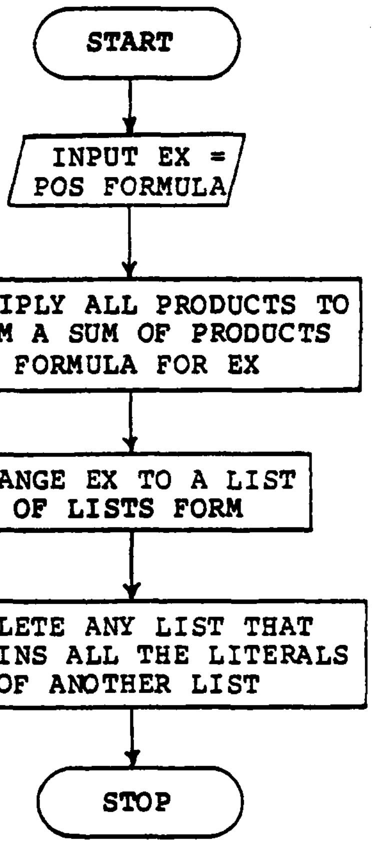

START
INPUT EX = POS FORMULA
MULTIPLY ALL PRODUCTS TO FORM A SUM OF PRODUCTS FORMULA FOR EX
CHANGE EX TO A LIST OF LISTS FORM
DELETE ANY LIST THAT CONTAINS ALL THE LITERALS OF ANOTHER LIST
STOP

Fig. 13. Routine to generate a sum of products formula from a product of sums formula (convertpos).

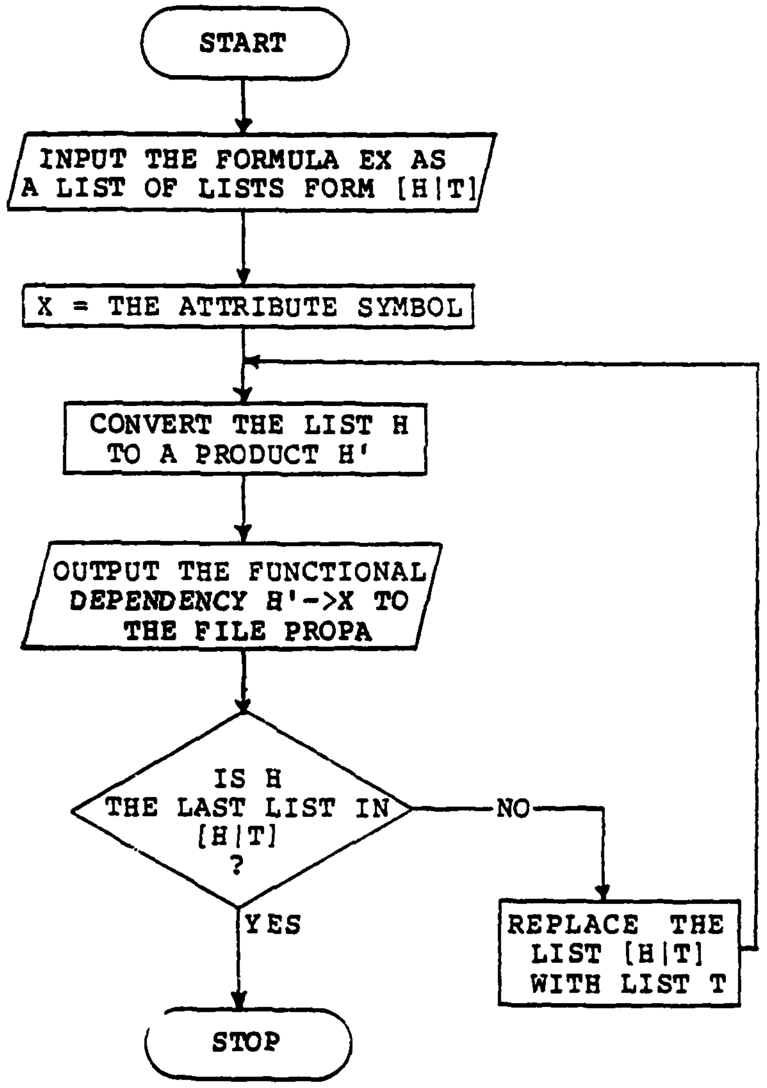
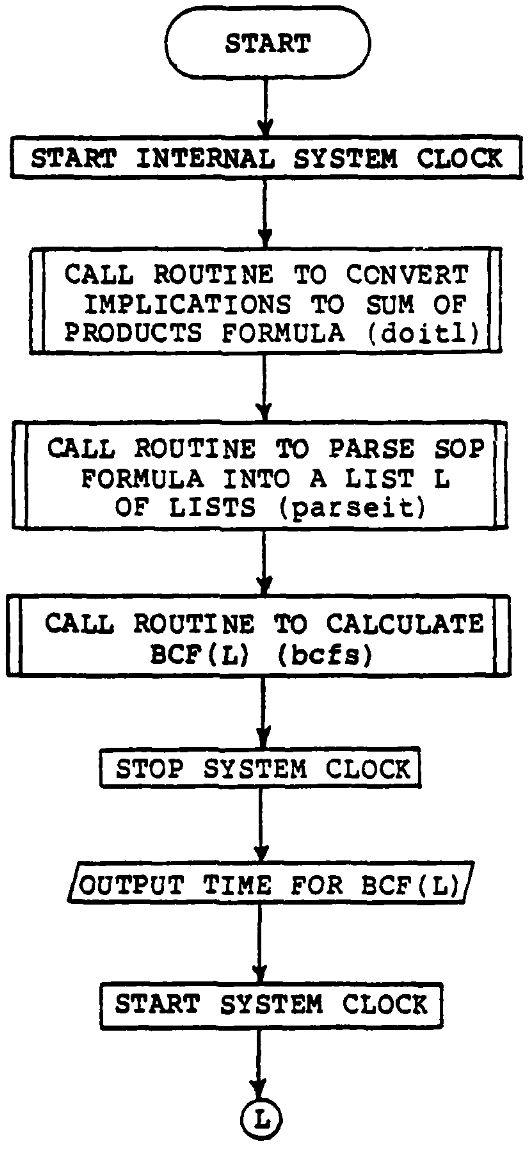
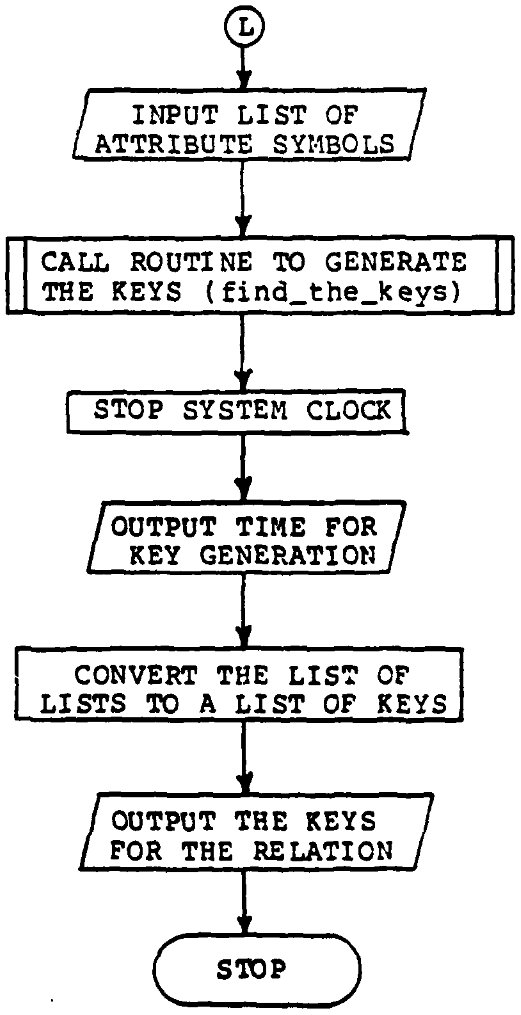

START
INPUT THE FORMULA EX AS A LIST OF LISTS FORM [H|T]
X = THE ATTRIBUTE SYMBOL
CONVERT THE LIST H TO A PRODUCT H'
OUTPUT THE FUNCTIONAL DEPENDENCY H'->X TO THE FILE PROPA
IS H THE LAST LIST IN [H|T] ?
NO
YES
REPLACE THE LIST [H|T] WITH LIST T
STOP

Fig. 14. Routine to generate the functional dependencies from a list of lists (formdeps).


START
START INTERNAL SYSTEM CLOCK
CALL ROUTINE TO CONVERT IMPLICATIONS TO SUM OF PRODUCTS FORMULA (doitl)
CALL ROUTINE TO PARSE SOP FORMULA INTO A LIST L OF LISTS (parseit)
CALL ROUTINE TO CALCULATE BCF(L) (bcfs)
STOP SYSTEM CLOCK
OUTPUT TIME FOR BCF(L)
START SYSTEM CLOCK
L

15a.

Fig. 15. Key generation routine (solve_for_keys).


L
INPUT LIST OF ATTRIBUTE SYMBOLS
CALL ROUTINE TO GENERATE THE KEYS (find_the_keys)
STOP SYSTEM CLOCK
OUTPUT TIME FOR KEY GENERATION
CONVERT THE LIST OF LISTS TO A LIST OF KEYS
OUTPUT THE KEYS FOR THE RELATION
STOP

15b.

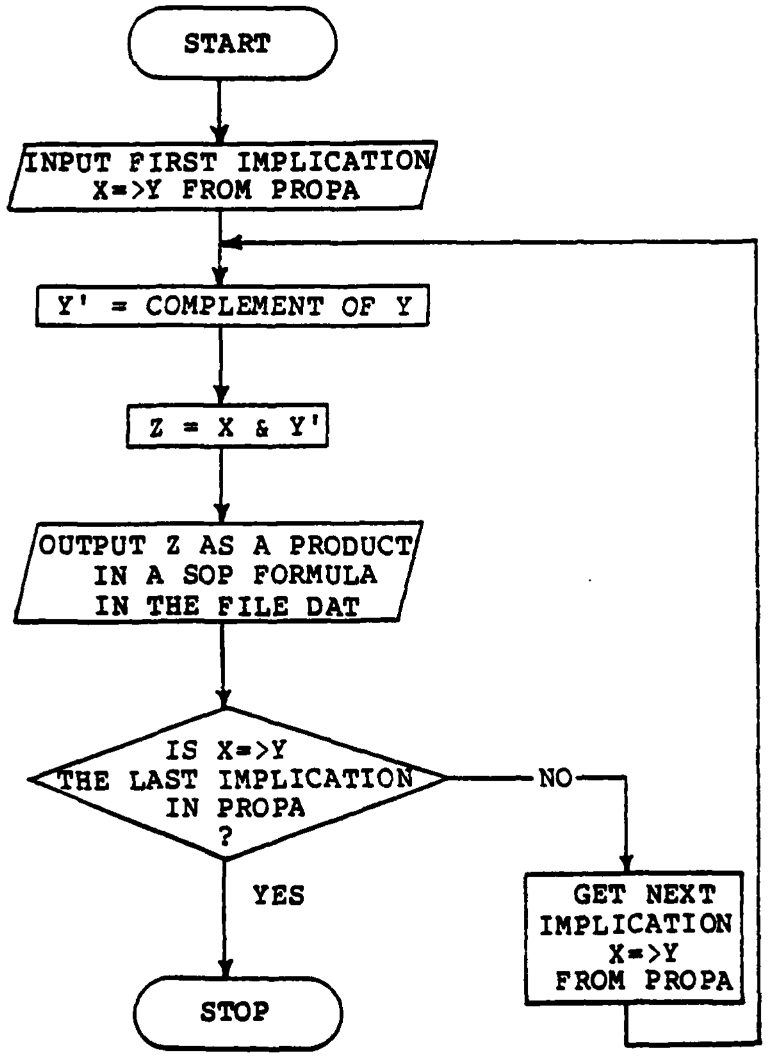
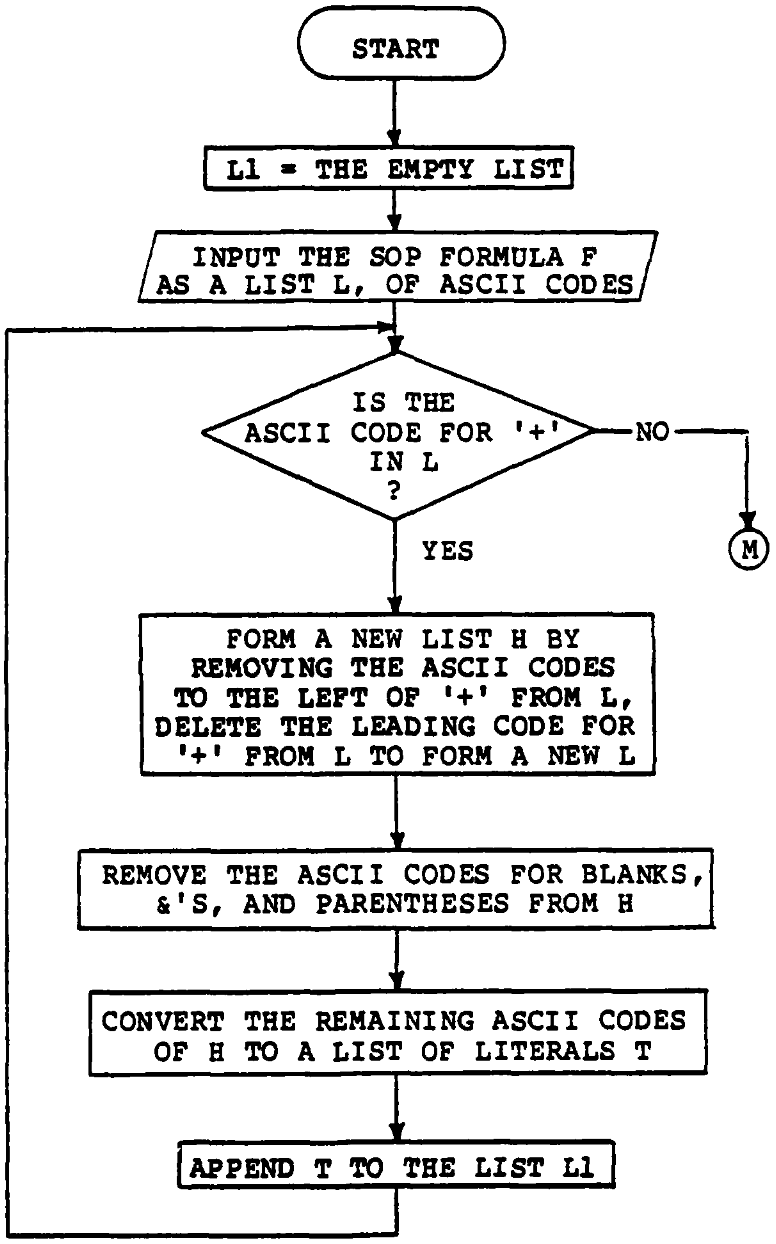
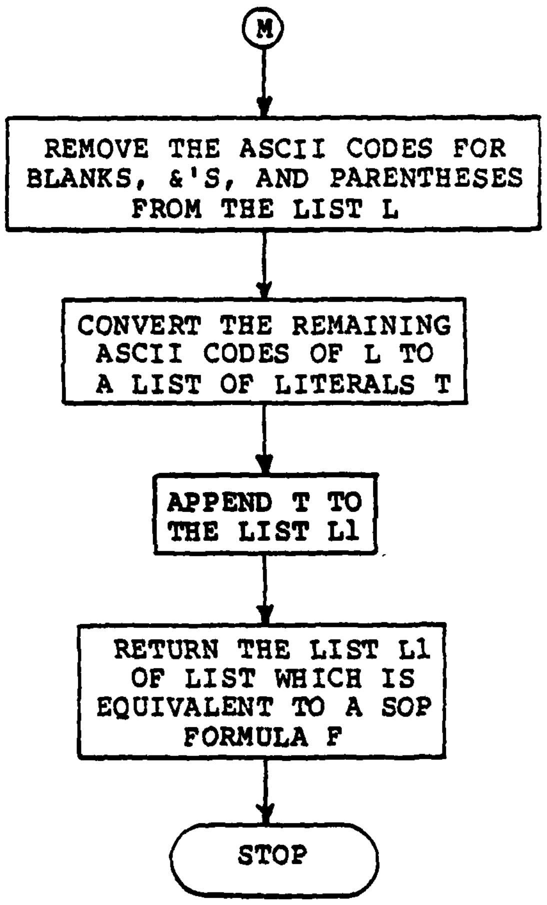

START
INPUT FIRST IMPLICATION X=>Y FROM PROPA
Y' = COMPLEMENT OF Y
Z = X & Y'
OUTPUT Z AS A PRODUCT IN A SOP FORMULA IN THE FILE DAT
IS X=>Y THE LAST IMPLICATION IN PROPA ?
NO
YES
GET NEXT IMPLICATION X=>Y FROM PROPA
STOP

Fig. 16. Routine to convert implications to sum of products formula (doitl).


START
L1 = THE EMPTY LIST
INPUT THE SOP FORMULA F AS A LIST L, OF ASCII CODES
IS THE ASCII CODE FOR '+' IN L ?
NO
YES
M
FORM A NEW LIST H BY REMOVING THE ASCII CODES TO THE LEFT OF '+' FROM L, DELETE THE LEADING CODE FOR '+' FROM L TO FORM A NEW L
REMOVE THE ASCII CODES FOR BLANKS, &'S, AND PARENTHESES FROM H
CONVERT THE REMAINING ASCII CODES OF H TO A LIST OF LITERALS T
APPEND T TO THE LIST L1

17a.

Fig. 17. Routine to parse a sum of products formula into a list of lists form (parseit).


M
REMOVE THE ASCII CODES FOR BLANKS, &'S, AND PARENTHESES FROM THE LIST L
CONVERT THE REMAINING ASCII CODES OF L TO A LIST OF LITERALS T
APPEND T TO THE LIST L1
RETURN THE LIST L1 OF LIST WHICH IS EQUIVALENT TO A SOP FORMULA F
STOP

17b.

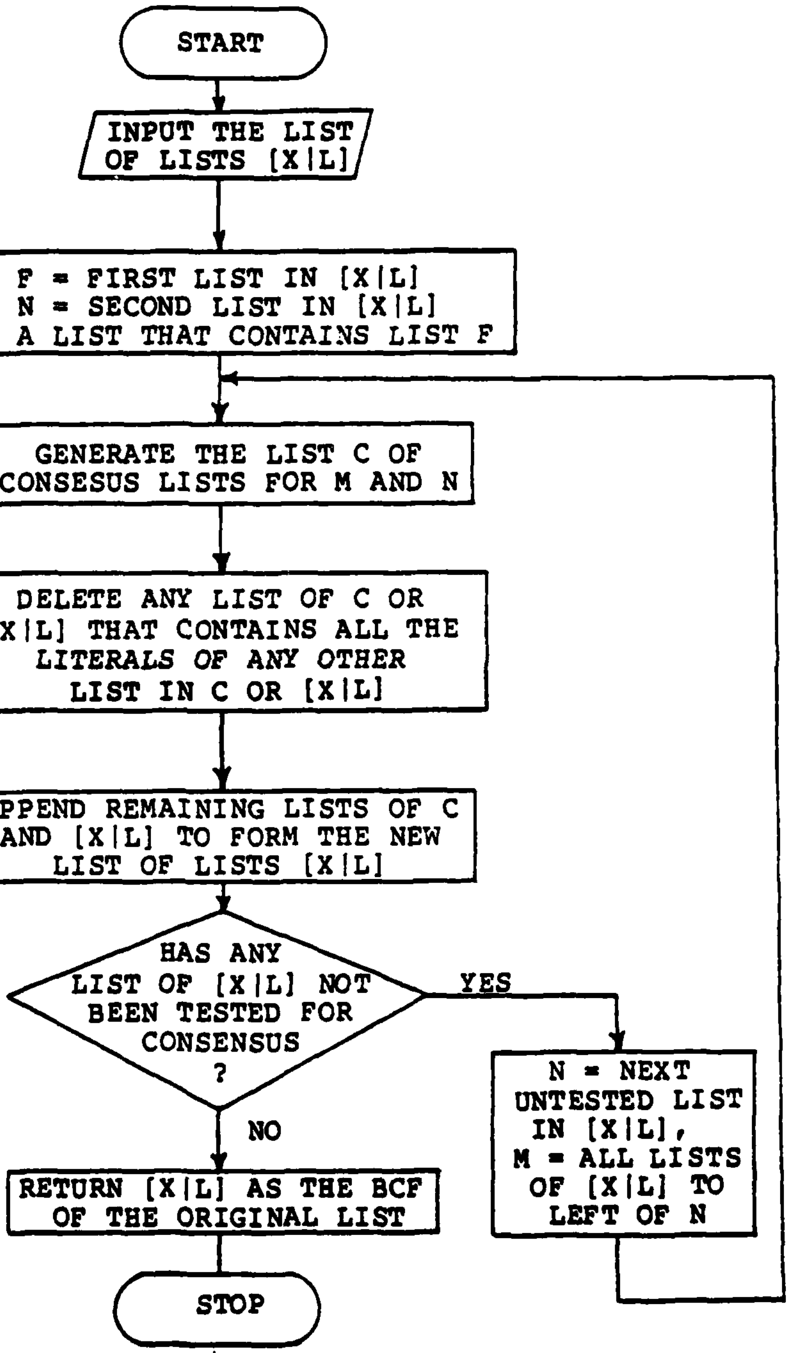

START
INPUT THE LIST OF LISTS [X|L]
F = FIRST LIST IN [X|L]
N = SECOND LIST IN [X|L]
M = A LIST THAT CONTAINS LIST F
GENERATE THE LIST C OF CONSESUS LISTS FOR M AND N
DELETE ANY LIST OF C OR [X|L] THAT CONTAINS ALL THE LITERALS OF ANY OTHER LIST IN C OR [X|L]
APPEND REMAINING LISTS OF C AND [X|L] TO FORM THE NEW LIST OF LISTS [X|L]
HAS ANY LIST OF [X|L] NOT BEEN TESTED FOR CONSENSUS ?
YES
N = NEXT UNTESTED LIST IN [X|L], M = ALL LISTS OF [X|L] TO LEFT OF N
NO
RETURN [X|L] AS THE BCF OF THE ORIGINAL LIST
STOP

Fig. 18. BCF routine for a sum of products formula F (bcfs).

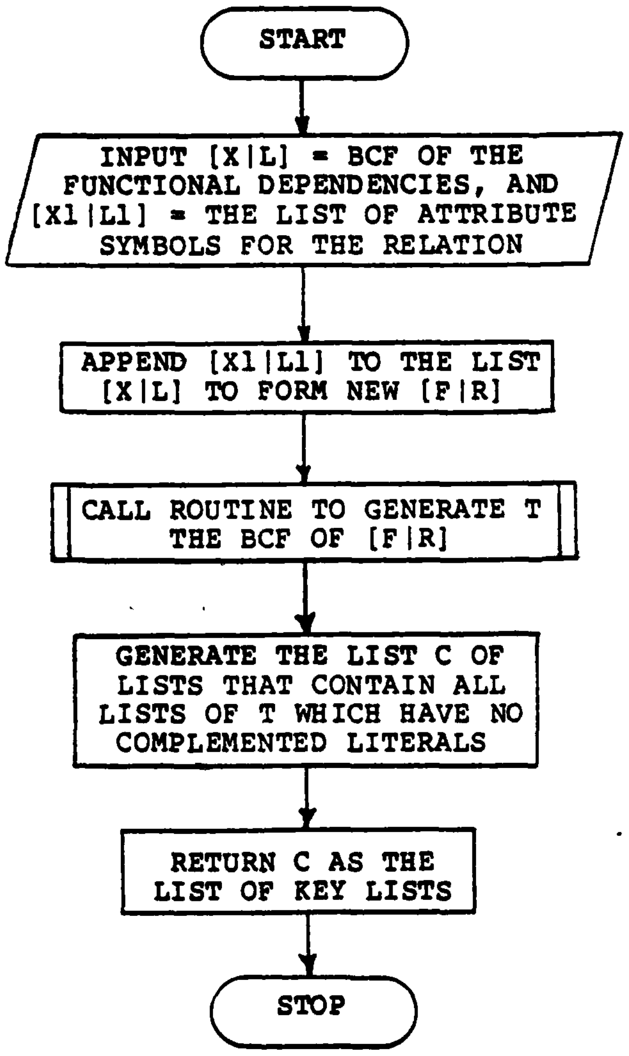

START
INPUT [X|L] = BCF OF THE FUNCTIONAL DEPENDENCIES, AND [X1|L1] = THE LIST OF ATTRIBUTE SYMBOLS FOR THE RELATION
APPEND [X1|L1] TO THE LIST [X|L] TO FORM NEW [F|R]
CALL ROUTINE TO GENERATE T THE BCF OF [F|R]
GENERATE THE LIST C OF LISTS THAT CONTAIN ALL LISTS OF T WHICH HAVE NO COMPLEMENTED LITERALS
RETURN C AS THE LIST OF KEY LISTS
STOP

Fig. 19. Routine to generate a list of key lists (find_the_keys).

APPENDIX B

PROLOG PROGRAM FOR THE FD-KEY ALGORITHM

```
:-public find_the_keys/3,keys/1.
:-public solve_for_keys/1,mainthing/1.

/************************************************/
/*                                              */
/* These are the operators: "+" is logic OR,"&"  */
/* is logic AND,                                */
/* "`" is logic NOT,and "=>" is                 */
/* logic implication.                           */
/*                                              */
/************************************************/

:-op(900,xfx,=>).
:-op(690,yfx,+).
:-op(600,yfx,&).
:-op(500,xf,`).

/************************************************/
/*                                              */
/* mainthing(A,A) is a routine to time the      */
/* functional dependency generation algorithm and*/
/* the key generation algorithm. mainthing also */
/* calls the routines to generate the functional */
/* dependencies and the keys for a relation.    */
/* A is the list of attribute symbols.          */
/*                                              */
/************************************************/

mainthing([X|L]):-time0(T),
  mainthing1([X|L],[X|L]),time0(T1),
  close(propa),Time is T1-T,
  write('Time for functional dependency '),
  write('generation is  '),
  nl,write(Time),write('ms'),nl,
  solve_for_keys([X|L]),clearfiles1.

/************************************************/
/*                                              */
/* clearfiles1 is a routine to delete the data  */
/* files dat, list, and blake.                  */
/*                                              */
/************************************************/

clearfiles1:-see(dat),rename(dat,[]),see(list),
  rename(list,[]),
  see(blake),rename(blake,[]).
```

```
/************************************************/
/*                                              */
/* mainthing1(A,A) is a routine to generate all */
/* of the functional dependencies of the relation*/
/* stored in the file dbase.  A is the list of  */
/* attribute symbols.                           */
/*                                              */
/************************************************/

mainthing1([],[P|Q]):-!.
mainthing1([X|L],[P|Q]):-setdatalist(X,[P|Q],[Z|K]),
  listofparts([Z|K],0,N),
  genblocks1(N,1,1,[H|T]),numattr(X,[P|Q],P1),
  formpartition([H|T],P1),removec(X,[P|Q],[H1|T1]),
  length([H|T],M),(formfunction([H1|T1],M,1,X),
  convertpos(X),formdeps(X);true),clearfiles(M,X),
  mainthing1(L,[P|Q]).


/************************************************/
/* setdatalist(X,L,L1,M) is a routine that      */
/* returns the list L1 of unique data values    */
/* found in the column X of the database        */
/* stored in the file dbase. The list L of      */
/* attributes for the database must be given    */
/* along with the length M of this list.  L     */
/* includes the attribute X.  This list must be */
/* in the same order as the columns of the dbase */
/************************************************/

setdatalist(X,[Y|L],[Z|K]):-numattr(X,[Y|L],N),
  see(dbase),
  setdata1(N,[H|T]),nodups([H|T],[Z|K]),seen.

/************************************************/
/* numattr(X,L,Z) is a routine that returns N the*/
/* numerical position of the attribute X in the */
/* list of attributes L.                        */
/************************************************/

numattr(X,[Y|L],N):-X=Y,N=1.
numattr(X,[Y|L],N):-numattr(X,L,N1),N is N1+1.

/************************************************/
/* setdata1(N,H) reads a tuple(row) from dbase  */
/* and returns H  all the data values found in  */
/* column N of the database.                    */
/*                                              */
/************************************************/
```

```
setdata1(N,P):-read(X),setdata2(N,X,P).

/**********************************************/
/* setdata2(N,B,C) determines if there are no    */
/* more rows in the database and returns C, an   */
/* empty list if this is true.  Otherwise, the   */
/* nth data value of the tuple B is placed into  */
/* the list of values C and another row is       */
/* obtained by setdata1.                         */
/**********************************************/

setdata2(N,end_of_file,[]):-!.
setdata2(N,X,[Y|T]):-nthinlist(N,X,Y,1),
  setdata1(N,T).

/**********************************************/
/* nodups(L,L1), removes any duplicate data      */
/* values from the list L and returns this       */
/* revised list L1.                              */
/*                                               */
/**********************************************/

nodups([],[]).
nodups([H|T],[Z|K]):-member(H,T),nodups(T,[Z|K]).
nodups([H|T],[H|K]):-nodups(T,K).

/**********************************************/
/* nthinlist(N,L,Y,M) returns the nth data value */
/* Y in the tuple L. M is a counter.             */
/**********************************************/

nthinlist(N,[X|L],X,M):-M=N.
nthinlist(N,[X|L],Y,M):-M1 is M+1,
  nthinlist(N,L,Y,M1).

/**********************************************/
/*                                               */
/* listofparts(J,N) inputs a listof unique data  */
/* values, and outputs N the number of partitions*/
/* of this data list stored in the files p1,...pN*/
/* where N is a number to be determined.         */
/*                                               */
/**********************************************/

listofparts(J,N,M):-length(J,Y),Y1 is Y+1,
  N1 is Y1/2,
  formblock1(0,N1,J,X,K1),testn(N,[X,K1],M).
```

```
/************************************************/
/*                                              */
/* listofparts1(M,[X|L],C) inputs a number N, a */
/* partition [X|L] in a list of lists form, and */
/* outputs a list of lists C which is a new     */
/* partition of [X|L].  Every list X in [X|L]   */
/* will have been separated into two lists and  */
/* inserted as two lists in the list of lists C.*/
/*                                              */
/************************************************/

listofparts1(N,[],[]):-!.
listofparts1(N,[H|T],[X1,K2|U]):-length(H,Y),
  Y1 is Y+1,
  N1 is Y1/2,formblock1(0,N1,H,X1,K2),N2 is N +1,
  listofparts1(N2,T,U).

/************************************************/
/*                                              */
/* formblock1(Z,N,L,L1,K) inputs the value of a */
/* counter Z, a list of data values L, and      */
/* outputs two new lists L1 and K which consist */
/* of a partition of the list L.  The number of */
/* of elements required to be in the list L1 is */
/* input as the number N.                       */
/*                                              */
/************************************************/

formblock1(Z,Z,K1,[],K1):-!.
formblock1(Z,N1,[J|K],[J|L],K1):-Z1 is Z+1,
  formblock1(Z1,N1,K,L,K1).

/************************************************/
/*                                              */
/* testn is a routine to call writepart.        */
/*                                              */
/************************************************/

testn(0,H,M):-writepart(0,H,M).
testn(N,[H|T],M):-writepart(N,[H|T],M).

/************************************************/
/*                                              */
/* writepart(N,L) is a routine that writes the  */
/* partition L in the file p(N+1).  N is an     */
/* input number and L is a list of lists.  This */
/* routine also calls testh(A,B).               */
/*                                              */
/************************************************/
```

```
writepart(N,H,Z):-N1 is N+1,name(N1,M),
  concat([112],M,P),
  name(P1,P),tell(P1),write(H),write('.'),
  told,testh(H,N1,Z).

/*************************************************/
/*                                               */
/* testh([H!T],N) inputs a number N and a        */
/* partition [H!T], which is in a list of lists  */
/* form.  N is the number associated with the file*/
/* pN where [H!T] is stored.  The length of the  */
/* list H is tested.  If the length is less than */
/* or equal to two, then the routine testt(T,N)  */
/* is called.  Otherwise, the N+1th partition is */
/* formed by calling listofparts1.               */
/*                                               */
/*************************************************/

testh([H!T],N,M):-length(H,Y),
  (Y=<2,testt(T,N,M);
  listofparts1(N,[H!T],G),testn(N,G,M)).

/*************************************************/
/*                                               */
/* testt(T,N) inputs a list of lists T and a     */
/* number N.  If T is an empty list the routine  */
/* succeeds, otherwise testh(T,N) is called.     */
/*                                               */
/*************************************************/

testt([],N,N):-!.
testt(T,N,M):-testh(T,N,M).

/*************************************************/
/*                                               */
/* genblocks(A,B,C,D) and genblocks1(A,B,C,D)    */
/* are routines to form a list D of two-block    */
/* partitions that are maximally skewed. A is    */
/* the number of partitions and both B and C are */
/* counters.                                     */
/*                                               */
/*************************************************/

genblocks1(Z,Z,M,[X]):-getname(Z,B),see(B),
  read(X),seen,test2(X).
genblocks1(Z,N,M,[X!L]):-getname(N,B),see(B),
  read(X),seen,genblocks(Z,N,M,L).
genblocks(Z,N,M,[X!L]):-getname(N,B),see(B),
  read(Y),seen,
  getapart(Z,N,1,T,Y),T=[X],(Z=N,L=[];N1 is N + 1,
  genblocks(Z,N1,1,L)).
genblocks(Z,N,M,L):-getname(N,B),see(B),
  read(X),seen,getapart(Z,N,M,L,X).

/*************************************************/
/*                                               */
/* test2([X|L]) is a routine to test if the      */
/* length of each list X in [X|L] is less than 2 */
/*                                               */
/*************************************************/

test2([]):-!.
test2([H|T]):-length(H,Y),!,Y<2,test2(T).

/*************************************************/
/*                                               */
/* getapart(A,B,C,D,E) returns E a maximally     */
/* skewed two-block partition.                   */
/*                                               */
/*************************************************/

getapart(Z,N,2,[],X):-!.
getapart(Z,N,M,[X1|L1],[X|L]):-M1 is M+1,
  getblock(Z,N,M1,[X1],X,L),
  N1 is N+1,getapart(Z,N1,M1,L1,L).

/*************************************************/
/*                                               */
/* getblock(A,B,C,D,E,F) and getblock1(A,B,C,D,E)*/
/* generate a block of the partition.            */
/*                                               */
/*************************************************/

getblock(Z,N,M,X,[H|T],[]):-length([H|T],Y),
  Y1 is Y+1,N2 is Y1/2,
  getblock1([H|T],0,N2,W,V),concat([W],[V],X).
getblock(Z,N,M,[X],[H|T],[H1|T1]):-
  length([H|T],Y),Y1 is Y+1,N2 is Y1/2,
  getblock1([H|T],0,N2,W,V),
  getblock(Z,N,M,[X1|L1],H1,T1),
  formsome([W,V],[X1|L1],X).
getblock1(H,N2,N2,[],H):-!.
getblock1(H,N,N,X,H):-!.
getblock1([H|T],C,N2,[H|T1],V):-C1 is C+1,
  getblock1(T,C1,N2,T1,V).

/*************************************************/
/*                                               */
/* getname(N,B) is a routine to form the file    */
/* name B=pN, where N is a number.               */
/*                                               */
/*************************************************/
```

```
setname(N,B):-name(N,M),concat([112],M,V),
  name(B,V),!.

/************************************************/
/*                                              */
/* formsome(A,B,C) forms a partition C from two */
/* blocks A and B.                              */
/*                                              */
/************************************************/

formsome([W,V],[X],Z):-formsome([W,V],X,Z).
formsome([W,V],[A,B],[X2,L2]):-concat(W,A,X2),
  concat(V,B,L2).

/************************************************/
/*                                              */
/* formpartition(X,Y) inputs a list X of Y two- */
/* block partitions. Each partition is a list   */
/* containing two lists or blocks.  This routine */
/* will form 2Y files such that each file       */
/* corresponds to a block in a partition.  Hence,*/
/* Y copies of the file dbase will be created.  */
/* These new relations will only contain data   */
/* values not found in the partitions.          */
/*                                              */
/************************************************/


formpartition([X|L],P):-see(dbase),
  formparts1([X|L],1,P),
  seen,told.

/************************************************/
/*                                              */
/* formparts1([X|L],R,P) performs two different */
/* tests on a row from the relation. Test 1     */
/* determines if the last row in the dbase has  */
/* been processed.  A true response will initiate*/
/* test 2. A false response will call the routine*/
/* part1. Test 2 determines if the last partition*/
/* has been processed.  A true response will    */
/* terminate the routine. A false response will */
/* increment the counter R and call formparts1. */
/*                                              */
/************************************************/
```

```
formparts1([X|L],R,P):-read(G),
  (G=end_of_file,(L=[];R1 is R+1,
  close(dbase),see(dbase),formparts1(L,R1,P));
  part1(X,[X|L],G,R,1,P)).
part1([Y|N],[X|L],[A|B],R,Q,P):-[Z|K]=Y,
  compare_the_value(Z,[A|B],P,1),
  name(R,F),name(Q,E),
  concat(F,E,W),concat([112],W,C),name(I,C),tell(I),
  writetuple(Z,[A|B],[H|T]),write([H|T]),write('.'),
  nl,formparts1([X|L],R,P).

/************************************************/
/*                                              */
/* part1(L,L1,L2,R,Q,N) places the tuple L2 from */
/* the file dbase into the file pRQ, which is the*/
/* Qth block of partition R. This is accomplished*/
/* by testing for membership of a data value in  */
/* block Q of partition R, in the tuple L2 and   */
/* writing this into the correct file.           */
/*                                              */
/************************************************/

part1([Y|N],[X|L],[A|B],R,Q,P):-length(Y,N1),N1=1,
  part1(N,[X|L],[A|B],R,2,P).
part1([Y|N],[X|L],[A|B],R,Q,P):-[Z|K]=Y,
  part1([K|N],[X|L],[A|B],R,1,P).

/************************************************/
/*                                              */
/* writetuple(Z,L,L1) returns a list L1 which is */
/* a subtuple of the tuple L with the data value */
/* Z removed.                                   */
/*                                              */
/************************************************/

writetuple(Z,[Z|B],B):-!.
writetuple(Z,[A|B],[A|K]):-writetuple(Z,B,K).

/************************************************/
/*                                              */
/* removec(X,L,L1) is a routine to remove the    */
/* attribute symbol X from the list of attribute */
/* symbols L, and return this new list L1.       */
/*                                              */
/************************************************/

removec(X,[],[]):-!.
removec(X,[X|L],L):-!.
removec(X,[H|T],[H|L1]):-removec(X,T,L1).
```

```
/***********************************************/
/*                                             */
/* compare_the_value(Z,L,N,C) determines if Z is */
/* the Nth data value in the tuple L, using the  */
/* counter C.                                  */
/*                                             */
/***********************************************/

compare_the_value(Z,[],N,N):-fail.
compare_the_value(Z,[Z|B],N,N):-!.
compare_the_value(Z,[A|B],N,Q):-
  N>Q,Q1 is Q + 1,!,
  compare_the_value(Z,B,N,Q1).

/***********************************************/
/*                                             */
/* formfunction(A,B,C,D) is a routine to form a  */
/* product of sums formula which contains all    */
/* information necessary to generate the         */
/* functional dependencies with attribute C on   */
/* the right side. This product of sums formula  */
/* is stored in the file EC where C is the       */
/* attribute symbol.  A is the list of ordered   */
/* attribute symbols with C removed. B is the    */
/* number of copies of the relation. D is a      */
/* counter used to access the correct copy of the*/
/* relation.                                   */
/*                                             */
/***********************************************/

formfunction(Q,M,N,C):-tell(temp),
  settuple1(N,X),
  settuple2(N,Z),
  formeqn(X,X,Z,Q,Q,M,N,C),!.

/***********************************************/
/*                                             */
/* settuple1(N,X) returns X a tuple of file pN1  */
/* if all the tuples have been read X = [ ].     */
/*                                             */
/***********************************************/

settuple1(N,X):-name(N,Z),name(1,Y),
  concat(Z,Y,W),concat([112],W,M),
  name(P,M),see(P),read(X1),
  (X1=end_of_file,X=[],seen;X=X1),!.
```

```
/***********************************************/
/*                                             */
/* settuple2(N,X) is identical to settuple1    */
/* except file pN2 is accessed.                */
/*                                             */
/***********************************************/

settuple2(N,Z):-name(N,X),name(2,Y),
  concat(X,Y,W),concat([112],W,M),
  name(P,M),see(P),read(Z1),
  (Z1=end_of_file,Z=[],seen;Z=Z1),!.

/***********************************************/
/*                                             */
/* formean(A,B,C,D,E,F,G,H) is a routine to test */
/* for non-identical data values in the tuples */
/* from pG1 and pG2. Originally A=B from pG1, C */
/* is a tuple from pG2, D and E are lists of   */
/* attribute symbols with the symbol H removed, */
/* and F is the number of data files.  If B=C, no*/
/* functional dependencies exist and formean will*/
/* fail. If the tuples are different, the first */
/* data values of each tuple are tested for    */
/* equality.  If they are equal, the rest of   */
/* tuple B will replace B and the rest of the  */
/* tuple C will replace C in the next call of  */
/* formean.  If the data values are different, */
/* the attribute symbol associated with these  */
/* values is stored as a literal in a sum in the */
/* file temp.  If there are no other data values */
/* in B and C, testtuple is called.  Otherwise */
/* formean2 is called.                         */
/*                                             */
/***********************************************/

formean(P,[X|L],[Z|K],Q,[A|B],M,N,C):-
  [X|L]=[Z|K],!,fail.
formean(P,[X|L],[Z|K],Q,[A|B],M,N,C):-X=Z,!,
  formean(P,L,K,Q,B,M,N,C),!.
formean(P,[X|L],[Z|K],Q,[A|B],M,N,C):-
  write(A),(L=[],
  settuple2(N,Z1),testtuple(P,X,Z1,Q,[A|B],M,N,C);
  formean2(P,L,K,Q,B,M,N,C)),!.
```

```
/*************************************************/
/*                                               */
/* formean2(A,B,C,D,E,F,G,H) is a routine similar*/
/* to formean.  The only differences are that A  */
/* is the comlete tuple of pG1, B and C are parts*/
/* of the tuples from pG1 and pG2, and E is the  */
/* associated part of the attribute list.  Also, */
/* if the subtuples B and C are equal, there will*/
/* be no more literals placed in the sum stored  */
/* in the file temp.                             */
/*                                               */
/*************************************************/

formean2(P,[X|L],[Z|K],Q,[A|B],M,N,C):-
  [X|L]=[Z|K],write('.'),nl,
  settuple2(N,Z1),testtuple(P,P,Z1,Q,Q,M,N,C),!.
formean2(P,[X|L],[Z|K],Q,[A|B],M,N,C):-X=Z,!,
  formean2(P,L,K,Q,B,M,N,C),!.
formean2(P,[X|L],[Z|K],Q,[A|B],M,N,C):-write('+'),
  write(A),(L=[],write('.'),nl,
  settuple2(N,Z1),testtuple(P,P,Z1,Q,[A|B],M,N,C);
  formean2(P,L,K,Q,B,M,N,C)),!.

/*************************************************/
/*                                               */
/* testtuple(A,B,C,D,E,F,G,H) is a routine to    */
/* test if there are no more tuples in the file  */
/* pG2. If this is true, testtuple1 is called,   */
/* otherwise formean is called with the new tuple*/
/* C from pG2.                                   */
/*                                               */
/*************************************************/

testtuple(P,X,Z,Q,A,M,N,C):-Z=[],settuple1(N,X1),
  testtuple1(X1,X1,Z,Q,Q,M,N,C),!.
testtuple(P,X,Z,Q,A,M,N,C):-
  formean(P,P,Z,Q,Q,M,N,C),!.

/*************************************************/
/*                                               */
/* testtuple1(A,B,C,D,E,F,G,H) is a routine to   */
/* test if there are no more tuples in pG1. If   */
/* this is true, the file temp is closed and the */
/* routine andtemp is called to generate the POS */
/* formula.  If this was the last data file, the */
/* routine changeit is called to delete any      */
/* extraneous sums in the POS formula. Then      */
/* formalph is called to form the logical AND of */
/* each formula generated by each data file. If  */
/* this was not the last data file, the routine  */
/* changeit is called to absorb an extraneous    */
/* sums in the POS formula and the next data file*/
/* is examined.                                  */
/*                                               */
/*************************************************/

testtuple1(P,X,Z,Q,A,M,N,C):-X=[],told,
  see(temp),read(Y),
  andtemp(N,Y),(N=M,changeit(N),
  formalph(M,C,e,1);N1 is N+1,changeit(N),
  formfunction(Q,M,N1,C)),!.
testtuple1(P,X,Z,Q,A,M,N,C):-
  settuple2(N,Z1),formean(P,P,Z1,Q,A,M,N,C),!.

/*************************************************/
/*                                               */
/* andtemp(N,X) inputs a sum of attribute symbols*/
/* Y from the file temp, and calls andtemp2.     */
/*                                               */
/*************************************************/

andtemp(N,end_of_file):-!.
andtemp(N,X):-setname(N,B),tell(B),write('('),
  write(X),write(')'),read(Y),andtemp2(N,Y),!.

/*************************************************/
/*                                               */
/* andtemp2(N,X) writes X as a product of the    */
/* POS formula stored in the file pN.            */
/*                                               */
/*************************************************/

andtemp2(N,end_of_file):-
  nl,seen,told,!.
andtemp2(N,X):-write('&'),write('('),
  write(X),write(')'),read(Y),andtemp2(N,Y),!.

/*************************************************/
/*                                               */
/* formalph(M,C,e,N) inputs the number of data   */
/* files M, the constant e, the attribute symbol */
/* C, and tests if the pN contains the last POS  */
/* formula.  If this is true, the formula is     */
/* stored in the file eC.  Otherwise X = the POS */
/* formula and formalph1(X,M,C,e,N) is called.   */
/*                                               */
/*************************************************/
```

```
formalph(M,C,e,N):-M=N,getname(N,V),see(V),
  read(X),seen,name(C,Z),name(e,K),
  concat(K,Z,B),name(L,B),tell(L),write(X),
  write('.'),told,!.
formalph(M,C,e,N):-getname(N,V),see(V),read(X),seen,
  N1 is N+1,formalph1(X,M,C,e,N1),!.

/***********************************************/
/*                                             */
/* formalph1(X,M,C,e,N) is a routine similar to */
/* formalph except that X is the list containing */
/* all of the POS formulas for the pN data files.*/
/* this final list is stored in eC if the last  */
/* formula is in X, otherwise formalph1 is called*/
/*                                             */
/***********************************************/

formalph1(Q,M,C,e,N):-M=N,getname(N,W),see(W),
  read(X),seen,concat(Q,X,Z),
  name(C,L),name(e,K),concat(K,L,B),
  name(D,B),tell(D),abspr(Z,[],F),
  write(F),write('.'),told,!.
formalph1(Q,M,C,e,N):-getname(N,W),see(W),
  read(X),seen,concat(Q,X,Z),
  N1 is N+1,formalph1(Z,M,C,e,N1),!.

/***********************************************/
/*                                             */
/* changeit(N) converts the POS formula in the  */
/* file pN to a list of list form by calling    */
/* nparse. Also, any extraneous lists are deleted*/
/* by the routine abspr.  This new list is stored*/
/* in the file pN again.                        */
/*                                             */
/***********************************************/

changeit(N):-name(N,Z),concat([112],Z,W),name(P,W),
  see(P),get(C),getstr(C,G),seen,nparse(F,G,[]),
  abspr(F,[],U),
  tell(P),write(U),write('.'),told,!.
```

```
/***********************************************/
/*                                             */
/* convertpos(C) inputs the list X of lists form */
/* of all the POS formulas stored in the file eC.*/
/* The list X has all extraneous lists absorbed  */
/* by the routine abspr. This new list Z is      */
/* stored by the as a POS formula Q in the file  */
/* eC.  This formula Q is translated into a SOP  */
/* formula by simp(Q,K). Then K is stored in the */
/* file eC.  The ascii code for K is input and   */
/* converted to a list W of lists form.  Then the*/
/* extra lists are absorbed by abspr(W,[],V).    */
/* Finally, the list V is stored in the file eC. */
/*                                             */
/***********************************************/

convertpos(C):-name(C,M),name(e,N),concat(N,M,P),
  name(L,P),see(L),read(X),
  seen,abspr(X,[],Z),tell(L),writeeq(Z),
  told,see(L),read(Q),seen,
  simp(Q,K),tell(L),write(K),told,
  see(L),get(S),getstr(S,N1),seen,
  parse(W,N1,[]),abspr(W,[],V),tell(L),write(V),
  write('.'),told,!.

/***********************************************/
/*                                             */
/* writeeq(A), writeeq1(A), writeeq2(A) inputs a */
/* list A of lists and stores a POS formula in   */
/* the file eC that corresponds to this list.    */
/*                                             */
/***********************************************/

writeeq([X|L]):-write('('),[Y|K]=X,write(Y),
  writeeq1(K),writeeq2(L),!.
writeeq1([]):-write(')'),!.
writeeq1([Y|K]):-write('+'),write(Y),writeeq1(K),!.
writeeq2([]):-write('.'),!.
writeeq2([X|L]):-write('&'),write('('),[Y|K]=X,
  write(Y),writeeq1(K),
  writeeq2(L),!.

/***********************************************/
/*                                             */
/* Nparse(Z) returns Z a list of lists for a POS */
/* formula.                                      */
/*                                             */
/***********************************************/
```

```
nparse(Z) --> nterm(X),"&",nrestparse(X,Z).
nparse([Z])-->nterm(Z).

/**************************************************/
/*                                                */
/* Nterm(Z) converts a term in the POS formula    */
/* into a list.                                   */
/*                                                */
/**************************************************/

nterm(Z) --> ndelim,ntoken(X),ndelim,nresterm(X,Z),!.

/**************************************************/
/*                                                */
/* Ndelim removes the following symbols from the */
/*  list created by nterm, &,+,`,(,).             */
/*                                                */
/**************************************************/

ndelim --> "+",!.
ndelim --> " ",!.
ndelim --> "(",!.
ndelim --> ")",!.
ndelim --> [],!.


/**************************************************/
/*                                                */
/* nresterm(X,Y) sets the first element of list   */
/* Y to the element                               */
/* X, and calls nterm to find the rest of the     */
/* list Y.                                        */
/*                                                */
/**************************************************/

nresterm(X,[X|R]) --> nterm(R).
nresterm(X,[X]) --> [].

/**************************************************/
/*                                                */
/* ntoken(Y) returns an atom Y for a member of    */
/* the ascii list.                                */
/*                                                */
/**************************************************/

ntoken(Y) --> [X],[96],{name(Y,[X,96])},!.
ntoken(Y) --> [X],{name(Y,[X]),Y\==’&’},!.
```

```
/**************************************************/
/*                                                */
/* nresparse(X,Y) sets the first element of the  */
/* list of lists Y to be the list X, which        */
/* corresponds to the first term of the POS       */
/* formula.  nrestparse then calls nparse.        */
/*                                                */
/**************************************************/

nrestparse(X,[X|R]) --> nparse(R).
nrestparse(X,[X]) --> [].


/**************************************************/
/*                                                */
/* formdeps(C) writes all the functional          */
/* dependencies with attribute C on the right     */
/* side that exist in the relation into the file */
/* propa.                                         */
/*                                                */
/**************************************************/

formdeps(C):-getlist(C,[X|L]),
  tell(propa),formdeps1(C,[X|L]).

/**************************************************/
/*                                                */
/* formdeps1(C,L) inputs an attribute C and a     */
/* list of lists L.  L is a list of all the left */
/* sides of the functional dependencies in the    */
/* relation.  This calls setterm and itself until*/
/* until L is an empty list.                      */
/*                                                */
/**************************************************/

formdeps1(C,[]):-!.
formdeps1(C,[X|L]):-setterm(C,X),formdeps1(C,L).

/**************************************************/
/*                                                */
/* setterm(C,L) inputs C the right side attribute*/
/* symbol and a list L of attributes for the left*/
/* side of a functional dependency.               */
/*                                                */
/**************************************************/

setterm(C,[Z|K]):-K=[],write(Z),
  write('=>'),write(C),write('.'),nl.
setterm(C,[Z|K]):-write(Z),write('&'),setterm(C,K).
```

```
/**************************************************/
/*                                                */
/* getlist(C,L) inputs an attribute C and returns*/
/* a list of left sides for functional           */
/* dependencies that was stored in the file EC.  */
/*                                                */
/**************************************************/

getlist(C,X):-name(C,G),concat([101],G,V),
  name(M,V),see(M),read(X),seen.

/**************************************************/
/*                                                */
/* clearfiles and nosubscripts are routines to   */
/* delete the data files used in the functional  */
/* dependency generation routine.                */
/*                                                */
/**************************************************/

clearfiles(N,X):-name(X,Z),concat([101],Z,W),
  name(W1,W),tell(W1),rename(W1,[]),
  tell(temp),rename(temp,[]),nosubscripts(N,1).
nosubscripts(N,M):-name(M,M1),
  concat([112],M1,M2),concat(M2,[49],M3),
  name(M4,M3),tell(M4),rename(M4,[]),
  concat(M2,[50],M5),name(M6,M5),
  tell(M6),rename(M6,[]),name(Q,M2),
  tell(Q),rename(Q,[]),(N=M;Z is M+1,
  nosubscripts(N,Z)).



/**************************************************/
/*                                                */
/* The SOP formula to be put in Blake Canonical  */
/* Form is represented as a list F of lists and  */
/* each of the lists in F corresponds to a term  */
/* in the SOP formula. . The method of iterated  */
/* consensus is used.  doitall is a procedure to */
/* time the routine for propositional logic to   */
/* BCF translation.  doit is a routine to read   */
/* in the propositional logic statements and     */
/* return the BCF of them.  doit1 converts the   */
/* propositions into a SOP form.  bcfs inputs a  */
/* list of lists and outputs the BCF of this list*/
/* as a list of lists.  parseit parses the SOP   */
/* formula into a list of lists.                 */
/*                                                  */
/* keys is  a routine to input the information      */
/* needed to locate the keys of a relation.  The    */
/* BCF form of the functional dependencies must     */
/* be stored in the file BLAKE. This routine        */
/* also prints out the time for the execution       */
/* and the keys. L is the attribute symbols.        */
/*                                                  */
/****************************************************/

solve_for_keys(L):-doitall,keys(L).

doitall:-time0(T),doit,time0(T1),Time is T1-T,
  write('Time  for bcf is   '),nl, write(Time),
  write('ms.'),nl.
doit:-see(propa),read(X),(compare(=,X,end_of_file);
  trans(X,Z),tell(dat),
  write(Z),doit1,nl,seen,told,parseit,bcfs).
doit1:-read(X),(compare(=,X,end_of_file);
  write('+'),
  nl,trans(X,Z),write(Z),doit1).
parseit:-see(dat),get(C),getstr(C,X),
  parse(P,X,[]),
  tell(list),write(P),write('.'),seen,told.
bcfs:-see(list),read(X),seen,bcf(X,Z),
  tell(blake),write(Z),write('.'),told.
keys(L):-time0(T),see(blake),read(Y),seen,
  find_the_keys(L,Y,N),time0(T1),
  Time is T1-T,write('Time for key search is '),
  write(Time),write(' ms.'),nl,write_keys1(N).
bcfit([X|L],[Z|K]):-time0(T),
  bcf([X|L],[Z|K]),time0(T1),
  Time is T1-T,write('TIME IS    '),nl,
  write(Time),write('ms').

/****************************************************/
/*                                                  */
/* bcf(A,B) returns  B = BCF(A).                    */
/*                                                  */
/****************************************************/

bcf([X|L],[Z|K]):-testit([],[X|L],M,N),
  bcf1([X|L],[M],N,[Z|K]).
```

```
/***********************************************/
/*                                             */
/* bcf1(A,B,C,D) generates a list of consensus   */
/* lists between the list C and the list of     */
/* lists B.  The list of consensus lists and the */
/* list B of lists are tested for absorptions.   */
/* The new left part of the list A along with the*/
/* next list of B is determined and bcf1 is      */
/* called again.  The BCF of the list A of lists */
/* is returned.                                  */
/*                                             */
/***********************************************/

bcf1([X|L],[M|R],[],[X|L]).
bcf1([X|L],[M|R],N,[Z|K]):-consc([M|R],N,C),
  abspr(C,[X|L],[Y|S]),
  (member(N,[Y|S]),start1(N,[Y|S],M1,N1),
  bcf1([Y|S],M1,N1,[Z|K]);
  reverse([M|R],[X2|L2]),
  testit([X2|L2],[Y|S],M1,N1),
  bcf1([Y|S],M1,N1,[Z|K])).

/***********************************************/
/*                                             */
/* abspr(A,B,C) checks for  absorptions between  */
/* the list A of                                 */
/* consensus lists and the old list of lists B.  */
/*  C is the new                                 */
/* list of lists formed after all absorptions.   */
/*                                             */
/***********************************************/

abspr([Y|S],[],[H|T]):-abspr(S,[Y],[H|T]).
abspr([],[X|L],[X|L]).
abspr([Y|S],[X|L],[Z|K]):-absp(Y,[X|L],[X1|L1]),
  abspr(S,[X1|L1],[Z|K]).

/***********************************************/
/*                                             */
/* start1(A,B,C,D) determines where the old next */
/* list A is in the new list B of lists, and     */
/* returns both the new next list D and the new  */
/* left part C of the list of lists.             */
/*                                             */
/***********************************************/
```

```
start1(N,[],[],[]).
start1(N,[Y|S],[M1],N1):-N=Y,
  M1=N,[N1|L]=S.
start1(N,[Y|S],[Y|L],N1):-start1(N,S,L,N1).

/************************************************/
/*                                              */
/* testit(A,B,C,D) determines if the first list */
/* of the list A of lists is in the new list B.  */
/* The new left part C and the new next term D  */
/* is returned.                                 */
/*                                              */
/************************************************/

testit([X2|L2],[Y|S],M1,N1):-member(X2,[Y|S]),
  start1(X2,[Y|S],M1,N1).
testit([X2|L2],[Y|S],M1,N1):-testit(L2,[Y|S],M1,N1).
testit([],[Y|S],Y,X):-[X|L]=S.
testit([],[Y],[Y],[]).

/************************************************/
/*                                              */
/* reverse(A,B) returns B the reverse order of A */
/*                                              */
/************************************************/

reverse([X],[X]).
reverse([X,Y],[Y,X]).
reverse([X|R],L):-re arse(R,L1),concat(L1,[X],L).

/************************************************/
/*                                              */
/* concat(A,B,C) returns the list C, comprised  */
/* of the list B appended to the list A.        */
/*                                              */
/************************************************/

concat([],L,L).
concat([F|L1],L2,[F|L3]):-
concat(L1,L2,L3).

/************************************************/
/*                                              */
/* getstr(A,B) returns a list of ascii          */
/* characters.                                  */
/*                                              */
/************************************************/
```

```
setstr(26,[]):-!.

setstr(C,[C|R]):-set(C2),setstr(C2,R),!.

/**********************************************/
/*                                            */
/* time is a routine to call the internal     */
/* timer of the system.                       */
/*                                            */
/**********************************************/

time0(T):-statistics(runtime,[T,_]).
time:-statistics(runtime,[_,T]),write(T),nl.


/**********************************************/
/*                                            */
/* neg(A,B) is a routine that returns the     */
/* complement B of a                          */
/*   boolean expression A.                    */
/*                                            */
/**********************************************/

neg(X&Y,A+B):-neg(X,A),neg(Y,B),!.
neg(X+Y,A&B):-neg(X,A),neg(Y,B),!.
neg(X`,Y):-Y=X,!.
neg(X,Y):-Y=X` ,!.

/**********************************************/
/*                                            */
/* simp(A,B) and mult(C,D) are routines that  */
/* return the sum of products form of a formula */
/* that is in product of sums form.           */
/*                                            */
/**********************************************/

simp(X&X,X):-!.
simp(X+X,X):-!.
simp(X+Y,Z):-simp(X,R),simp(Y,S),(Z)=(R+S) ,!.
simp(X&Y,Z):-simp(X,R),simp(Y,S),mult(R,S,Z),!.
simp(X,X):-!.
mult(X,X,X):-!.
mult(A+B,C+D,Z):-mult(A,C+D,Y),
  mult(B,C+D,X),(Z)=(X+Y),!.
mult(A&B,A,A&B):-!.
mult(A&B,B,A&B):-!.
mult(A,A&B,A&B):-!.
mult(B,A&B,A&B):-!.
mult(A+B,C,Z):-mult(A,C,X),mult(B,C,Y),(Z)=(X+Y),!.
mult(C,A+B,Z):-mult(C,A,X),mult(C,B,Y),(Z)=(X+Y),!.
mult(X,Y,X&Y):-!.
```

```
/**************************************************/
/*                                                */
/* trans(A,B) is a routine that translates a      */
/* propositional logic statement A into a SOP     */
/* formula B.                                     */
/*                                                */
/**************************************************/

trans(V=>Y,Z):-neg(Y,W),simp(V&W,Z).

/**************************************************/
/*                                                */
/* absp(A,B,C) returns C, a list of lists         */
/* corresponding to a SOP  formula.  C is the     */
/* result of performing absorption on the SOP     */
/* formula B with the list A.  A is a list and    */
/* B is a list of lists.                          */
/*                                                */
/**************************************************/

absp([],[X|L],[X|L]):-!.
absp(A,[],[A]):-!.
absp(A,[X|L],[X|L]):-sublist(X,A),!.
absp(A,[X|L],[Z|R]):-sublist(A,X),absp(A,L,[Z|R]),!.
absp(A,[X|L],[X|R]):-absp(A,L,R),!.

/**************************************************/
/*                                                */
/* sublist(A,B) determines if the list A is       */
/* contained in the list B.                       */
/*                                                */
/**************************************************/

sublist([X|L],[Y|S]):-member(X,[Y|S]),
  sublist(L,[Y|S]),!.
sublist([],M):-!.

/**************************************************/
/*                                                */
/* member(A,B) determines if the element A is     */
/* contained in the list B.                       */
/*                                                */
/**************************************************/

member(X,[X|R]):-!.
member(X,[Y|R]):-member(X,R),!.
member([],X):-!.
```

```
/**************************************************/
/*                                                */
/* consc(A,B,C) returns a list C of lists         */
/* consisting of all consensus terms between the */
/* list A and every list contained in the list B */
/* of lists. List C consists of non-empty lists. */
/*                                                */
/**************************************************/

consc([X|L],Y,[Z|K]):-cons([X|L],Y,[M|L1]),
  delete([M|L1],[Z|K]).

/**************************************************/
/*                                                */
/* delete(A,B) removes any empty lists contained */
/* in the list of lists A and returns the list of*/
/* lists B which is void of any empty lists.     */
/*                                                */
/**************************************************/

delete([M|L1],[Z|K]):-(compare(=,L1,[[]]),
  delete1(M,[Z|K]);
  compare(=,L1,[]),[Z|K]=[M|L1];
  Z=M,delete(L1,K)).
delete1(M,[M]).

/**************************************************/
/*                                                */
/* cons(A,B,C) returns a list C of lists          */
/* consisting of all consensus terms between the */
/* list A and every list contained in the list B */
/* of lists. The list C may contain empty lists. */
/*                                                */
/**************************************************/

cons([],Y,X):-[]=..X.
cons([X|L],Y,[M|L1]):-testcons1(X,Y,Y,Z),
  (compare(=,Z,[]),cons(L,Y,[M|L1]),!;
  compare(=,Z,[[]]),!,cons(L,Y,[M|L1]);
  M=Z,cons(L,Y,L1)).
cons([X|L],Y,[Z|K]):-cons(L,Y,[Z|K]).
```

```
/************************************************/
/*                                              */
/* testcons1(A,B,C,D) tests for a literal in    */
/* opposition between the list A and C.  D is the*/
/* returned consensus term.  B and C are        */
/* are identical when testcons1 is originally   */
/* called.  If no literal in opposition is found,*/
/* D is set to the empty list.  If a literal in */
/* opposition is found testcons2(V,W,X,Y,Z)     */
/* is called.                                   */
/*                                              */
/************************************************/

testcons1([A|C],Y,[],[]).
testcons1([A|C],Y,[B|D],X):-sublist([B],[A|C]),
  testcons1([A|C],Y,D,X).
testcons1([A|C],Y,[B|D],X):-neg(B,G),
  sublist([G],[A|C]),
  testcons2([A|C],Y,D,[B,G],X).
testcons1([A|C],Y,[B|D],X):-testcons1([A|C],Y,D,X).

/************************************************/
/*                                              */
/* testcons2(A,B,C,D,E) tests for a second      */
/* literal in opposition between list A and C.  */
/* Lists A and B are the original lists, list D */
/* contains the first literal in opposition and */
/* its complement, and list E contains the      */
/* consensus term.  If a second literal in      */
/* opposition is found , E is the empty list.   */
/*                                              */
/************************************************/

testcons2([A|C],Y,[],[B,G],[Z|K]):-
  formit1([A|C],Y,[B,G],[Z|K]).
testcons2([A|C],Y,[E|L],[B,G],X):-neg(E,F),
  sublist([F],[A|C]),[] =..X.
testcons2([A|C],Y,[E|L],[B,G],[Z|K]):-
  testcons2([A|C],Y,L,[B,G],[Z|K]).

/************************************************/
/*                                              */
/* formit1(A,B,C,D) places all non-opposition   */
/* literals in list B into the consensus list D. */
/* List C contains the opposition literal and   */
/* its complement.                              */
/*                                              */
/************************************************/
```

```
formit1([A|C],[],[B,G],[Z|K]):-.
  formit2([A|C],[B,G],[Z|K]).
formit1([A|C],[D|E],[B,G],[Z|K]):-B=D,
 formit1([A|C],E,[B,G],[Z|K]).
formit1([A|C],[D|E],[B,G],[Z|K]):-
  sublist([D],[A|C]),
  formit1([A|C],E,[B,G],[Z|K]).
formit1([A|C],[D|E],[B,G],[D|K]):-.
  formit1([A|C],E,[B,G],K).

/**********************************************/
/*                                            */
/* formit2(A,B,C,D) places all non-opposition */
/* literals of list A into the consensus list C. */
/* List B contains the opposition literal and */
/* its complement.                            */
/*                                            */
/**********************************************/

formit2([],[B,G],[]).
formit2([A|C],[B,G],X):-G=A,formit3(C,X).
formit2([A|C],[B,G],[A|K]):-formit2(C,[B,G],K).

/**********************************************/
/*                                            */
/* formit3(A,B) sets the tail of A of a list to */
/* be the list B.                             */
/*                                            */
/**********************************************/

formit3(C,C).

/**********************************************/
/*                                            */
/* parse(Z) returns Z  a list of list from a  */
/* SOP formula.                               */
/*                                            */
/**********************************************/

parse(Z) --> term(X),"+",restparse(X,Z).
parse([Z])-->term(Z).

/**********************************************/
/*                                            */
/* term(Z) converts a term in the SOP formula */
/* into a list .                              */
/*                                            */
/**********************************************/
```

```
term(Z) --> delim,token(X),delim,resterm(X,Z),!.

/***********************************************/
/*                                             */
/* delim removes the following symbols from the */
/* list created by term(Z), blanks, &, +, `, (,).*/
/*                                             */
/***********************************************/

delim --> "&",delim,!.
delim --> " ",delim,!.
delim --> "(",delim,!.
delim --> ")",delim,!.
delim --> [],!.

/***********************************************/
/*                                             */
/* resterm(X,Y) sets the first element of list Y */
/* to be the element X, and calls term to find  */
/* the rest of the list Y.                     */
/*                                             */
/***********************************************/

resterm(X,[X!R]) --> term(R).
resterm(X,[X]) --> [].

/***********************************************/
/*                                             */
/* token(Y) returns an atom Y for a member of  */
/* ascii list.                                 */
/*                                             */
/***********************************************/

token(Y) --> [X],[96],{name(Y,[X,96])},!.
token(Y) --> [X],{name(Y,[X]),Y\=='+'},!.

/***********************************************/
/*                                             */
/* restparse(X,Y) sets the first element of the */
/* list of lists Y to be the list X, which     */
/* corresponds to first term of the SOP formula. */
/* restparse then calls parse.                 */
/*                                             */
/***********************************************/

restparse(X,[X!R]) --> parse(R).
restparse(X,[X]) --> [].
```

```
/***********************************************/
/*                                             */
/* find_the_keys(A,B,C) inputs a list A of     */
/* symbols representing all of the attributes  */
/* for a relation in a database, and a list B  */
/* which corresponds to the BCF of the functional*/
/* dependencies of the relation.  The list C   */
/* contains the list of keys for the relation. */
/*                                             */
/***********************************************/

find_the_keys([X1|L1],[X|L],[F|R]):-
  bcf1([X|L],[X|L],[X1|L1],[Z|K]),
  determine([Z|K],[F|R]).

/***********************************************/
/*                                             */
/* determine(A,B) inputs the list A of lists   */
/* corresponding to the Blake Canonical Form of */
/* the functional dependencies and list of     */
/* attribute symbols for a relation.  The list */
/* of keys B is returned.                      */
/*                                             */
/***********************************************/

determine([],[]).
determine([Z|K],N):-(negsin(Z,M),
  N=[X|L],X=M,determine(K,L);
  determine(K,N)).

/***********************************************/
/*                                             */
/* negsin(A,B) inputs a list A of lists and    */
/* outputs a list B of lists that contain all  */
/* the lists of A that have no complemented    */
/* literals as members.  The list B is a list of */
/* keys of varying length.                     */
/*                                             */
/***********************************************/

negsin([],[]).
negsin([X|L],[H|T]):-[X]=[M`],!,fail.
negsin([X|L],[X|T]):-negsin(L,T).
```

```
/***************************************************/
/*                                                 */
/* write_keys1(A), write_keys(A), write_keys2(B) */
/* output the keys and a heading for them.         */
/*                                                 */
/***************************************************/

write_keys1([H|T]):-write('The keys are: '),
  nl,write_keys([H|T]).

write_keys([]):-nl.
write_keys([H|T]):-write_keys2(H),write_keys(T).
```

APPENDIX C

EXECUTIONS OF THE FD-KEY ALGORITHM

```
.type dbase
[a1,b1,c1,d1].
[a2,b3,c1,d1].
[a1,b2,c2,d2].
[a1,b1,c1,d2].
[a2,b3,c1,d2].
[a1,b2,c2,d1].

.run prolog

Prolog-10  version 3

[ Consulting 'prolog.ini' ]
| ?- restore(cfd).

[ closing all active files ]

[ restore complete ]

yes
| ?- mainthing([a,b,c,d]).
Time for functional dependency generation is
4251ms
Time for bcf is
172ms.
Time for key search is 331 ms.
The keys are:
dca
db

yes
yes
| ?- halt.

[ Prolog execution halted ]

EXIT

.type propa
b=>a.
c&a=>b.
b=>c.
```

```
.type dbase
[smith,surgery,4,b2,6].
[jones,pathology,2,a1,3].
[jones,pathology,1,c3,5].
[evans,anatomy,2,c3,10].
[jones,pathology,2,a1,7].
[evans,surgery,3,a2,3].
[smith,anatomy,5,a1,5].

.run prolog

Prolog-10  version 3

[ Consulting 'prolog.ini' ]
| ?- restore(cfd).

[ closing all active files ]

[ restore complete ]

yes
| ?- mainthing([p,c,y,r,t]).
Time for functional dependency generation is
17148ms
Time for bcf is
15418ms.
Time for key search is 16090 ms.
The keys are:
rt
yt
pt
ct


yes
| ?- halt.

[ Prolog execution halted ]

EXIT
```

```
.type propa
t&c=>p.
t&y=>p.
t&r=>p.
y&c=>p.
y&r=>p.
c&r=>p.
t&p=>c.
t&r=>c.
t&y=>c.
r&p=>c.
r&y=>c.
y&p=>c.
p&t=>y.
p&r=>y.
c&t=>y.
c&r=>y.
r&t=>y.
p&t=>r.
p&y=>r.
c&t=>r.
c&y=>r.
y&t=>r.
```

```
.type propa
a=>b.
c=>b&d.
b=>e&f.
e=>d.
f=>a&d.

.run prolog

Prolog-10  version 3

[ Consulting 'prolog.ini' ]
| ?- restore(cfd).

[ closing all active files ]

[ restore complete ]

yes
| ?- solve_for_keys([a,b,c,d,e,f]).
Time for bcf is
1432ms.
Time for key search is 5971 ms.
The keys are:
c


yes
| ?- halt.

[ Prolog execution halted ]

EXIT
```

REFERENCES

1. Aho, A. V., Y. Sagiv, and J. D. Ullman, "Efficient Optimization of a Class of Relational Expressions," *ACM Transactions on Database Systems*, vol. 4, no. 4, pp. 435-454, December 1979.
2. Beeri, C., and P. A. Bernstein, "Computational Problems Related to the Design of Normal Form Relational Schemas," *ACM Transactions on Database Systems*, vol. 4, no. 1, pp. 30-79, March 1979.
3. Beeri, C., P. A. Bernstein, and N. Goodman, "A Sophisticate's Introduction To Database Normalization Theory," *Proceedings of The 4th International Conference on Very Large Databases*, West Berlin, pp. 113-124, September 1978.
4. Bernstein, P. A., "Synthesizing Third Normal Form Relations from Functional Dependencies," *ACM Transactions on Database Systems*, vol. 1, no. 4, pp. 277-298, December 1976.
5. Bernstein, P. A., and N. Goodman, "What Does Boyce-Codd Normal Form Do?", *Procedings of The 6th International Conference on Very Large Databases*, pp. 245-259, 1980.

6. Blake, A., "Canonical expressions in Boolean algebra," Dissertation, University of Chicago, Dep't. of Mathematics, August 1937, (cited in [8]).

7. Brown, F. M., EE 682 Lecture Notes, University of Kentucky, 1980.

8. Brown, F. M., "Inferential Processor," Final Report, 1980 USAF-SCEEE Summer Faculty Research Program, August 1980.

9. Champine, G. A., "Current Trends in Data Base Systems," Computer, p. 27, May 1979.

10. Codd, E. F., "A Relational Model of Data for Large Shared Data Banks," Communications of the ACM, vol. 13, no. 6, pp. 377-387, June 1970.

11. Coelho, H., J. C. Cotta, and L. M. Pereira, How to Solve It With Prolog, 2nd ed., Laboratório Nacional de Engenharia Civil, Lisboa, Portugal, 1980.

12. Date, C. J., An Introduction to Database Systems, 2nd ed., Reading, Mass.: Addison-Wesley Publishing Company, Inc., 1977.

13. Delobel, C., and R. G. Casey, "Decomposition of a database and the theory of Boolean switching functions," IBM Journal of Research and Development, vol. 17, pp. 374-386, September 1973.

14. Fagin, R., "The Decomposition Versus The Synthetic Approach to Relational Database Design," Proceedings of The 3rd International Conference on Very Large Databases, pp. 441-446, October 1977.

15. Gotlieb, C. C., and Leo R. Gotlieb, Data Types and Structures, Englewood Cliffs, N. J.,: Prentice-Hall, Inc., 1978.

16. Kohavi, Z., Switching and Finite Automata Theory, 2nd ed., New York, N. Y.,: McGraw-Hill Book Company, 1978.

17. Ling, Tok-Wang, F. W. Tompa, and T. Kameda, "An Improved Third Normal Form for Relational Databases," ACM Transactions on Database Systems, vol. 6, no. 2, pp. 329-346, June 1981.

18. Martin, James, Computer Data-base Organization, Englewood Cliffs, N. J.: Prentice-Hall, Inc., 1975.

19. Nicolas, J. M., "Mutual Dependencies and Some Results on Undecomposable Relations," *Proceedings of The 4th International Conference on Very Large Databases*, West Berlin, pp. 360-367, September 1978.

20. Pereira, L. M., F. Pereira, and D. H. D. Warren, "User's Guide to DECsystem 10 PROLOG," 1978.

21. Sagiv, Y., C. Delobel, D. S. Parker, and R. Fagin, "An Equivalence Between Relational Database Dependencies and a Subclass of Propositional Logic," To appear in *Journal of ACM*.

22. Warren, D. H. D., "Implementing PROLOG-Compiling Predicate Logic Programs," vol. 1 & 2-DAI, Research report no. 39, University of Edinburgh, 1977, (cited in [11]).

23. Wiederhold, G. O., *Database Design*, New York, N. Y.: McGraw-Hill Book Company, 1977.